Distributed Systems: Architecture-Driven Specification using Extended LOTOS

Ashley M^{c}Clenaghan

Department of Computing Science and Mathematics
University of Stirling, Scotland
September 1993

This thesis has been submitted to the University of Stirling in partial fulfilment of the requirements for the Degree of Doctor of Philosophy.

Abstract

The thesis uses the LOTOS language (ISO International Standard ISO 8807) as a basis for the formal specification of distributed systems. Contributions are made to two key research areas: **architecture-driven specification** and **LOTOS language extensions**.

The notion of architecture-driven specification is to guide the specification process by providing a reference-base of pre-defined domain-specific components. The thesis builds an infra-structure of architectural elements, and provides **Extended LOTOS (XL)** definitions of these elements.

The thesis develops Extended LOTOS (XL) for the specification of distributed systems. XL is LOTOS enhanced with features for the formal specification of quantitative timing, probabilistic and priority requirements. For distributed systems, the specification of these 'performance' requirements, can be as important as the specification of the associated functional requirements.

To support quantitative timing features, the XL semantics define a global, discrete clock which can be used both to force events to occur at specific times, and to measure intervals between event occurrences. XL introduces *time-policy* operators *ASAP* ('as soon as possible' corresponding to "maximal progress semantics") and *ALAP* ('as late as possible'). Special internal transitions are introduced in XL semantics for the specification of probability. Conformance relations based on a notion of *probabilization*, together with a testing framework, are defined to support reasoning about probabilistic XL specifications. Priority within the XL semantics ensures that permitted events with the highest priority weighting of their class are allowed first.

Both functional and performance specification play important rôles in CIM (Computer Integrated Manufacturing) systems. The thesis uses a CIM system known as the CIM-OSA Integrating Infrastructure as a case study of architecture-driven specification using XL.

The thesis thus constitutes a step in the evolution of distributed system specification methods that have both an architectural basis and a formal basis.

Declaration

I hereby declare that this thesis has been composed by myself, that the work reported has not been presented for any university degree before, and that the ideas that I do not attribute to others are due to myself.

Ashley M^c^Clenaghan

Ashley M^{c}Clenaghan
September 1993

Acknowledgements

This thesis would not have been completed without the help and support of many people. I would like to thank each of them even if I do not mention them by name.

I am indebted to Prof. Kenneth J. Turner, my supervisor, for creating the opportunity for this Ph.D., and for his constant encouragement, patience and guidance. I thank Ken for the substantial time, effort and expertise that he has generously contributed to my supervision. The LOTOS specifications in this thesis have been formatted using Ken's *lottex* program and *.sty* files.

I would like to thank Paul Gibson for his friendship and support, and for the competition.

The Department of Education for Northern Ireland (DENI) financed this Ph.D. work through a CAST award (Co-operative Award in Science and Technology). Hewlett-Packard Laboratories (Bristol) provided industrial sponsorship. I thank Hewlett-Packard for giving me the opportunity to work at their research laboratories in Bristol. In particular I would like to thank Stewart Black, Claudio Condini and Paul Mellor from Hewlett-Packard, for their help and supervision. I would like to thank the AMICE Consortium for allowing me to work on Esprit Project 688 CIM-OSA, and for providing financial assistance towards travelling expenses. In particular I would like to thank Patrick Viollet and Bruno Querenet (Hewlett-Packard, Villefontaine) for their help involving the CIM-OSA Project.

The Computing Science Department at Stirling has been an excellent place in which to work. I thank the department for its financial support, and all in the department for their friendship and help. Special mention goes to the Computer Support Group (Sam, Graham and Catherine) for their expertise and technical advice.

I thank Mr. Charles Rattray, Prof. David W. Bustard and Prof. Peter F. Linington for agreeing to be my Ph.D. examiners. I would like to thank Charles for also being my second supervisor.

I would like to mention some of my friends who have helped make my stay at Stirling unforgettable: Paul, Andrew, Nige, Dave, Claire, Liz, Ana, Sharon, Vivienne, Geoff, Andy, Julie, Grant, Alannah, Ian and Gary.

I especially want to thank my father, mother, Clive and Rosemary (my brother and sister) for all their support and encouragement. Thanks!

Contents

Chapter 1

Introduction

Thesis:

> Distributed systems can be effectively specified and analysed in an architecture-driven way using an extended form of the LOTOS formal specification language.

This chapter introduces the thesis. We begin with a guided tour through the areas of research that constitute the context of the thesis, indicating their relevance. Then we outline the extent of the thesis, and précis the research contributions made by the thesis. Finally, an overview of the structure of the thesis provides chapter-by-chapter navigation.

1.1 The context

This section overviews the context of the thesis. It provides a guide through the areas of research visited by the thesis, and indicates their relevance.

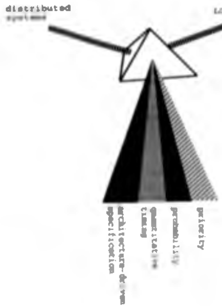


Figure 1.1: Research themes

1.1.1 Distributed systems

The area in computing science known as **distributed systems** [Hal88, CD88, Tan81, SK87] has rapidly grown in importance over the last two decades. This growth has been fueled by the falling costs of building networked systems and by the advancement of enabling technology. Distributed systems hold great potential. They promise computational power and speed through distributed processing, flexibility through the interconnection of diverse systems, and integration through communication. However, the potential of distributed computing systems is matched by their complexity.

Broadly speaking, the thesis develops specific intellectual tools for tackling key aspects of the complexity of distributed systems.

1.1.2 The need for architecture-driven formal development

Distributed computing systems are among the most complex constructions ever devised by humans. During the last few decades computing science has invested in the development of **formal** (mathematical) **languages** as a means of handling the complexities of designing distributed systems. However, demands for the accelerated production of distributed systems, coupled with increased complexity, mean that formal languages are by themselves not sufficient description tools for development of distributed systems. To meet the twin problems of productivity and complexity, this thesis forecasts that in the future, distributed systems will be specified, designed and built using pre-designed domain-specific components. Distributed system components may be less concrete than the components found in the manufacturing industries, but the goals of knowledge and resource re-use are the same.

The idea of providing and using pre-designed domain-specific components is incorporated in the notion of **architecture-driven** development. The phrase *architecture-driven* reflects the architectural basis of the components (see [Tur87, Tur91, Tur90, VSvS88, Bog90, Bie89, BB86, Pir91, Got92]).

1.1.3 Architecture-driven specification

The primary emphasis of the thesis is the **specification** of distributed systems, i.e. the description of *what* systems can do, not *how* they do it. The aim of **architecture-driven specification** is to guide the specification process by providing a reference-base of pre-defined domain-specific components.

1.1.4 Formality

Formality supports accurate specification and analysis of distributed systems. Formal languages are languages that are wholly defined in terms of axioms and inference rules, such as those of logic and set theory. Examples of formal languages include: LOTOS [ISO89b], SDL [CCI92], Estelle [ISO89a], CSP [Hoa85, Hoa85], CCS [Mil80], Z [Spi89], VDM [BJ78], and Petri Nets [Pet62, Pet81]. Unlike natural language which has no agreed set of definitions, a formal language enjoys objective interpretation. The preciseness of formal languages make them ideal as notations in the contract-like[1] world of systems specification.

Formal languages are precise, (relatively) concise, consistent and analysable; their use encourages specifications that are correct and complete with respect to the system requirements. A formal language is not, by itself, a solution to all of the traditional problems associated with the development of distributed systems. However, its qualities do go some way to alleviating the difficulties, and a formal notation together with formal reasoning may result in the automation of parts of the development process.

The thesis embraces formality in its use of the formal language LOTOS as a basis for the specification of distributed systems.

1.1.5 LOTOS

1988 saw ISO (the International Standards Organisation) grant International Standard status (ISO 8807) to a formal language known as **LOTOS** [ISO89b]. LOTOS is based on algebraic methods: a process algebra derived from Milner's CCS [Mil80]

[1] A specification may be considered a contract to be fulfilled by an implementation (see [WBL90]).

and Hoare's CSP [Hoa83]; and an abstract data type algebra, inspired by ACT ONE [EM85]. LOTOS was originally conceived as a specification language for OSI (Open Systems Interconnection [ISO84, BS81]), but the suitability of LOTOS for modelling a wide variety of discrete event distributed systems has now been recognized (e.g. [McC91a, Bie89, BB86, TS93]).

The basic concept of the process part of the language is describing the (observable) behaviour of a system in terms of the relative ordering of its actions. The LOTOS language has features for supporting abstraction, providing modularity, modelling concurrent behaviour, indicating synchronous behaviour, denoting non-determinism, representing spontaneous transitions and describing data structures. The net effect of these language features is to make LOTOS particularly good for capturing abstract descriptions of distributed, concurrent, non-deterministic systems.

A large knowledge-base of LOTOS know-how has evolved (see section 3.3.3) — this includes standards, tutorial literative, theoretical support, methods, example applications, and LOTOS related projects. Also, a number of supporting software tools are available (see section 3.4.3). These include syntax directed editors, syntax checkers, parsers, static semantics checkers, animators, verifiers, and transformation tools including compilers.

These attributes are among the reasons for choosing LOTOS as the most suitable language upon which to base the work in the thesis.

1.1.6 Reference architectures

Distributed systems are complex to specify and design. When engineers in other disciplines, such as civil engineering or electronic engineering, are faced with complex design tasks they consult their discipline's design guides for knowledge. Recognizing the importance of this paradigm, computing science has begun to establish a number of design guides for various sub fields within its discipline. Reference architectures for distributed systems include: **OSI** (Open Systems Interconnection) [ISO84, BS81], **ODP** (Open Distributed Processing) [vG89, Lin91, Ste91, ISO89e], **DAF** (Distributed Applications Framework) [CCI88a, CCI88c, CCI88b], **ANSA/ISA** (Advanced Networked Architecture/Integrated Systems Architecture) [ANS89b, ANS89a, ANS86], **ROSA** (RACE Open Service Architecture) [ROS89b, ROS89c, ROS89a, ROS89d], **ODA** (Office Document Architecture) [ISO88a, ISO88b] and **CIM-OSA** (Computer Integrated Manufacturing – Open Systems Architecture) [CIM90d, CIM90a, CIM89c].

Reference architectures support architecture driven specification. Reference architectures guide the specifier by: providing appropriate concepts and concise terminology for talking about the design space; structuring the design domain by partitioning problems and separating concerns to make the domain easier to understand; pre-defining generic components that can be customised, or common components that can be re-used; imparting domain knowledge and expertise that has been evolved by previous designers.

Reference architectures are relevant to the thesis on two levels. Firstly, in support of the notion of architecture-driven specification, the thesis builds its own *generic reference architecture* for distributed systems. This provides an infra-structure of formalised architectural concepts and components (chapter 4). Secondly, developing a specific reference architecture for CIM-OSA (Computer Integrated Manufacturing – Open Systems Architecture) has prompted some of the work in the thesis, and provided case-study material (chapter 5).

1.1.7 CIM-OSA

CIM OSA (Computer Integrated Manufacturing – Open Systems Architecture) [CIM90d, CIM90a, CIM89c, Bee89, McC91a] is a reference architecture for CIM (Computer Integrated Manufacturing) systems [ESP88, JBD89]. The CIM-OSA Reference Architecture defines concepts, generic structures and guidelines that can be used to integrate manufacturing and business elements of an enterprise within a information technology framework.

Attempts within the CIM-OSA project to formalise aspects of the CIM-OSA Reference Architecture involved the author [McC91a, McC90b, McC90a, MBB90]. It led him to recognise both the weakness of LOTOS for the specification of performance concerns (quantitative timing, probability, priority), and the need for an architectural framework to guide specification. To address the first weakness, the thesis extends LOTOS for the specification of performance (chapters 6, 7 and 8), and uses certain parts of CIM OSA as a case-study (chapter 5) to demonstrate the power of Extended LOTOS. To address the second concern, the thesis develops an infra-structure of formalised architectural elements (chapter 4) and uses this for the architecture driven specifications in the CIM OSA case-study of chapter 5.

1.2 The extent of the thesis

The two main products of the thesis are intellectual and architectural insight, and practical contributions to CIM-OSA development. These mark the extent of the thesis.

The thesis develops extensions to the LOTOS language and theory for reasoning about *performance* aspects of distributed systems. The thesis does not develop software tools to support these extensions.

LOTOS has been used to formalise parts of the CIM-OSA Reference Architecture [Vio90]. The author has contributed to this work [McC91a, McC90b, McC90a, MBB90]. Some of the ideas for Extended LOTOS have been prompted by CIM OSA work involving the author. Extended LOTOS has not officially been used within the project, although the need for *performance* extensions to LOTOS for specifying CIM systems has been officially recognised [Vio90]. The case study (chapter 5) of the application of Extended LOTOS to the specification of the CIM-OSA SE (System-Wide Exchange), is not work which has been carried out within the CIM-OSA project, although it is based on work by the author [McC90b] officially recognised by the CIM-OSA project.

The work in the thesis builds on existing research where possible. Where this has been used, appropriate acknowledgements are given.

1.3 Contributions of the thesis

This section summarizes the main research contributions made by the thesis.

The thesis contributes to three particular research areas:

- enhancements to the LOTOS language
- architectural concepts and their definition
- applications of the LOTOS language.

1.3.1 Enhancements to the LOTOS language

For distributed systems, *performance* (or, more generally, Quality of Service — QoS) concerns have an important status as well as functional concerns.[2] LOTOS is good at expressing functional aspects (such as behaviour, organizational structure and data-structures) of distributed systems, but inadequate for expressing performance-oriented aspects (such as quantitative timing, probability and priority). To remedy this inadequacy, the thesis proposes and formally defines three extensions of the LOTOS language:

TLOTOS: TLOTOS is LOTOS extended for the formal specification of quantitative timing concerns. TLOTOS has the following characteristics:

- TLOTOS semantics define a global, discrete clock which supports the notion of physical clocks [Lam78].
- Time values and operations are presented as pre-defined ACT ONE library types.
- Quantitative times can be absolute (relative to the global clock), or relative (to other events).
- Events can be forced to occur at specific times.
- Intervals between event occurrences can be measured.
- TLOTOS introduces *time-policy* operators: **ASAP** ('as soon as possible' [BL91]) requests an application of 'maximal progress semantics'[3]; while, **ALAP** requests the use of the 'as late as possible' policy.
- TLOTOS semantics ensure the sensible interleaving (in quantitative time) of events in parallel processes.
- TLOTOS specifications can be tested under extended definitions of the LOTOS testing relations.

PbLOTOS: PbLOTOS is LOTOS extended for the formal specification of probabilistic concerns. The PbLOTOS work makes the following contributions:

[2] Chapter 2 describes how, especially in distributed systems, performance concerns have a substantial impact on functional and "*correctness*" concerns.

[3] which state that, if there is nothing to prevent an event from occurring, then it must occur without delay

- Two derivatives of an LTS (Labelled Transition System) [Plo81] are defined: an NP-LTS which contains both non-deterministic and probabilistic transitions; and a P-LTS which contains only probabilistic transitions.
- NP-LTSs are used as a semantic model for PbLOTOS systems.
- PbLOTOS is defined to include a probabilistic choice operator for specifying probability distributions over a set of internal probability transitions. The probabilistic choice operator, together with the ability to express non-determinism, gives PbLOTOS the power to generate NP-LTSs.
- The thesis explains how an NP-LTS can be considered to be a *specification* which describes a set of P-LTS *implementations*. On this basis, an implementation relation (a pre-order) [Led91a], called *probabilization* is defined.
- The thesis gives an operational definition of *probabilization* and describes how this relation may be used to reason about PbLOTOS systems, and used as a notion of conformance in the development of probabilistic systems.
- A statistical testing framework is suggested for establishing whether a real-world (probabilistic) implementation is a valid implementation of a PbLOTOS specification, according to the probabilization relation.

PrLOTOS: PrLOTOS is LOTOS extended for the formal specification of priority concerns. PrLOTOS provides the following features:

- A prioritized event is given a *priority-class* and *priority-value*.
- Where there is a choice between events from the same priority-class, the event with the highest priority-value will be fired. A choice between events from different priority-classes is rationalized to a non-deterministic choice between the events with the highest priority-value in their respective priority-class.
- A choice between unprioritized events (events without explicit priority values) and prioritized events (events with explicit priority values) gives rise to non-deterministic choice.
- Prioritized event offers may synchronize with unprioritized event offers, prioritizing these unprioritized event offers through, what the thesis calls, *association*.

The thesis describes how the integration of TLOTOS, PbLOTOS and PrLOTOS produces **Extended LOTOS (XL)** — a formal, LOTOS based, specification language for the specification of distributed systems.

1.3.2 Architectural concepts and their definition

The thesis develops a 'method' for the application of LOTOS, called architecture-driven specification (see section 1.1.2). Architecture driven specification methods are advantageous because they re-use domain knowledge — know how built from a previous history of solutions. This domain knowledge is often embodied in the forms of generic concepts, ingredients, template-components, etc. The thesis develops an infra-structure of architectural concepts and components to support the architecture-driven specification of distributed systems.

This infra-structure is organized as a pyramid, with the fundamental elements at the base, and *common architectural components* at the apex. This set of components includes common *performance components*, as well as *functional components*, in recognition of the importance of the performance concerns in distributed systems. Components are given XL templates and graphical representations.

1.3.3 Applications of the LOTOS language

To capture and accumulate experience in the use of LOTOS, it is important to apply LOTOS to new problem domains. In part, this thesis reports on the application of LOTOS to aspects of the CIM-OSA Reference Architecture. CIM-OSA is interesting because it contrasts with other domains: CIM-OSA is not a symmetric, layered communications architecture such as OSI; and CIM-OSA is more applied and specialized than the very general ODP architecture. The thesis contains a CIM-OSA case-study which provides an insight into the advantages of architecture-driven specification. The case-study also illustrates how to use the special features of XL, and demonstrates the practical value of XL.

1.4 Thesis structure

This section provides a chapter-by-chapter guide to the thesis.

Chapter 2 introduces distributed systems as a context for the application of the work in the thesis. It examines both the theoretical/academic perceptions and industrial/application perceptions of distributed systems, and provides background material on specific systems that are relevant to this thesis. Finally, it highlights key areas of distributed systems research for the thesis.

Chapter 3 provides general background information about specification languages. It focuses on LOTOS and briefly looks at the factors involved in using LOTOS for system development. It concludes by highlighting key aspects of LOTOS language research for the thesis.

Chapter 4 builds an infra-structure of architectural elements to support the architecture-driven specification of distributed systems. Architectural elements are given XL representations and graphical notations. The work in this chapter provides a basis for the case-study in chapter 5. This chapter uses XL features defined in chapters 6, 7 and 8.

Chapter 5 is a case-study of architecture-driven, formal specification using XL. The chapter introduces the IIS (Integrating Infrastructure) — the part of CIM-OSA which was subjected to formal specification. The chapter uses chapter 4's infrastructure of architectural elements to construct a skeleton architecture of the IIS. Then the chapter shows how the *common architectural components*, defined in chapter 4, can be customized for the specification of both the functional requirements and performance requirements of a specific part of the IIS, known as SE (System Wide Exchange).

Chapter 6 defines extensions to LOTOS for the specification of quantitative timing concerns. The result is called TLOTOS. The chapter examines the inadequacies of LOTOS with respect to quantitative timing, it investigates the language features needed for the expression of quantitative timing, and it formally defines syntactic and semantic extensions to LOTOS for realising these facilities. The chapter also takes at look at ways of mapping TLOTOS to LOTOS.

Chapter 7 defines extensions to LOTOS for the specification of probabilistic concerns. The result is called PbLOTOS. The chapter extends the definition of LTSs (Labelled Transition Systems) to include both probabilistic and non-deterministic transitions. Extended LTSs are used as a semantic model for PbLOTOS. The chapter defines an pre-order relation called *probabilization* for PbLOTOS. The chapter shows how the probabilization can be used as a notion of conformance in the development of probabilistic systems. The chapter concludes by outlining a framework for the statistical testing of real world (probabilistic) implementations against PbLOTOS specifications.

Chapter 8 comes in two parts. The first part defines extensions to LOTOS for the specification of priority concerns. The result is called PrLOTOS. The second part of the chapter describes how TLOTOS, PbLOTOS and PrLOTOS integrate to form Extended LOTOS (XL). A simple example XL specification is provided to illustrate the expressive flexibility of XL for the specification of performance concerns.

Chapter 9 summarizes the thesis and identifies possibilities for further work.

Appendix A contains XL specifications that reflect the architecture driven decomposition of a CIM OSA IIS X_ACCP_Client_PS_SP component. This is reference material for section 5.3.6.

Appendix B contains the XL specification of the CIM OSA IIS SE component. This is reference material for section 5.4.1.

Appendix C contains ACT ONE data types for inclusion in the pre-defined data types library of TLOTOS. This is reference material for section 6.5.1.

Appendix D contains an example application of the TLOTOS semantics. This is reference material for section 6.5.4.

Appendix E contains an example which illustrates difficulties of representing aspects of the semantic mechanisms of TLOTOS in the syntax of LOTOS. This is reference material for section 6.6.2.5.

Appendix F contains a series of XL specifications which describe alternative designs (concerned with timing abstractions) for two parts of the CIM OSA IIS: the X_Service and X_Service_Agent. These XL specifications are used as reference material for sections 5.5, 6.2 and 6.7.

Appendix G forms an annex to chapter 6. This appendix extends the definitions of some of the LOTOS testing relations, and shows that these testing relations yield sensible and intuitive results when applied to TLOTOS specifications.

Appendix H provides an example of the application of the *SimChar* algorithm. The *SimChar* algorithm, defined in section 7.4.7, is used to give an operational definition to the notion of ***probabilization*** (chapter 7).

Appendix I lists abbreviations used in the thesis.

The work reported in chapters 4 and 5 prompted aspects of the develop of XL. Chapters 4 and 5 also provide an introduction by example to XL. This is the reason for ordering the work in chapters 4 and 5 before the work in chapters 6, 7 and 8, although chapters 4 and 5 rely on the definitions of XL provided in chapters 6, 7 and 8.

Chapter 2

An overview of distributed systems

This chapter introduces distributed systems as a context for the application for the work in this thesis. We begin by examining the theoretical/academic perceptions of distributed systems, and enumerate the features that characterize a distributed system. Then, we shift to industrial/application perceptions of distributed systems, and précis a selection of architectures and applications that are relevant to our work. Finally, we highlight the key areas within distributed systems research that are addressed by this thesis.

2.1 Introduction

Theoretical research in the area of distributed systems is aimed at the general problem of developing methods for the specification and design of distributed systems that manage their inherent complexity. This is the context of the work of this thesis.

This chapter is purely a context setting chapter. It concentrates on describing those aspects of the research area that are relevant to, or have influenced our work. The chapter does not contrast our work with existing work. Instead, we detail and contrast existing work where appropriate throughout chapters 4 to 8.

2.2 Fundamental aspects of distributed systems

In this section we examine the theoretical/academic perceptions of distributed systems, and enumerate the features that characterize a distributed system.

2.2.1 Spatial separation and concurrency

In theoretical terms, distributed systems include features such as concurrency, asynchronous communications, spatially distributed components, etc. The task of designing distributed systems is not an easy one because of the complexity which results from the interplay of such features.

One consequence of this interplay is the problem of establishing a total ordering[1] of events in a distributed system.

Usually we consider it possible to decide the total ordering of a set of events which occur within a confined space, provided that the spatial dimensions are such that the transmission delay between event sites within this space is negligible compared to the time between event occurrences in this space. This scenario is reminiscent of a single node (maybe a single or tightly coupled processor system) in a network.

However, the total ordering for any one node is actually only a partial ordering for the entire distributed system. A distributed system consists of many such nodes, and the transmission delay between nodes is not negligible compared to the time between event occurrences in a single node. This may make it impossible to establish a satisfactory total ordering for events in a distributed system.

The other factor which makes it difficult to establish event orderings is the explicit introduction of *parallelism*. A system may be able to execute a number of processes in parallel. Usually it will be possible to establish a total ordering for the events of any one of these processes. However, it is often impossible to do so for all the events when considering the system as a whole.

Events or processes are said to be *concurrent* if it is impossible to establish a total ordering of events due to *spatial separation* and/or explicit *parallelism*. It can be difficult to distinguish between concurrency existing due to spatial separation and concurrency existing due to explicit parallelism. But there is a distinction.

[1]We say that event x is ordered before event y if x can causally affect y (see [Lam78])

2.2.2 Definition

The term "concurrent system" is often used to describe systems where concurrency arises due to explicit parallelism (only, not spatial separation). The term "distributed system" is often used to describe systems where concurrency arises due to spatial separation and explicit parallelism. We have chosen "distributed systems" as the context for our thesis because our work involves solving problems that arise due to both spatial separation and explicit parallelism. (As a warning note, terminology is at best blurred in this area, and can confuse.)

The total ordering question manifests itself in a number of well known distributed system problem areas, e.g. replicated distributed databases, mutual exclusion of resources, fair scheduling in distributed systems, etc. The total ordering question is fundamental to many distributed systems problems. This is the basis for our "academic" definition of a distributed system:

> A distributed system is a system in which the transmission delay of messages, between spatially separate computing elements, is not negligible compared to the time between computing event occurrences in any computing element.

By varying what we mean by a "not negligible" transmission delay, we can use this definition of a distributed system to identify a range of systems. By relaxing the notion of a "not negligible" transmission delay, we identify "tightly-coupled" distributed systems, such as array processor systems, shared-memory systems, multi-processors, neural networks, etc. In contrast, by emphasising the notion of a "not negligible" transmission delay, we identify "loosely-coupled" distributed systems, such as workstation/server models, telecommunications networks, computer integrated manufacturing systems, etc.

2.3 Distributed systems in practice

In this section we look at industrial/application perceptions of distributed systems, and précis a selection of architectures/applications that are relevant to the work in this thesis.

We are primarily interested in "reference architectures", rather than specific distributed systems, and this is reflected in the selection of distributed systems overviewed in this section. Our aim here is to provide a flavour of the distributed system reference architectures that have influenced and provide a context for the work within this thesis.

2.3.1 Perceptions

The industrial/application perception of a distributed system is an application that spans multi-vendor, multi domain, heterogeneous computer networks. From the industrial/application perspective, distributed systems are defined by characteristics such as different failure modes, dynamic configuration, concurrent access, asychronous interactions, remote access, heterogeneous components, management of replication, migrating elements, federated management, security, performance and reliability.

Product solutions and research solutions to distributed systems include: OSI, ODP, ANSA/ISA, CIM-OSA and related standards work (see following subsections); distributed operating systems (such as Amoeba, Mach and Locus [CD88, TvR85]); distributed file systems (such as Sun NFS, XDFS and CFS [CD88, NH82, LPS81]); add-on features (such as Remote Procedure Call facilities [BN84, Nel81, WB87] and remote execution [Eur88]); application programmer interfaces (APIs) (such as X/Open, SVID and POSIX); and total solutions (such as Grapevine [SBN84], Chorus [RM87, Pec92]).

Industry recognizes the increasing importance for strategic IT including system integration and open systems. It is essential that these tasks are carried out within some architectural framework which ought to be generic (multi-domain), optimizable and vendor independent. Examples of architectural frameworks include: OSI, ODP, ANSA/ISA and CIM-OSA. An architectural framework consists of: functional and structural abstractions; non-constructively specified design templates; design rules of necessary components and structures; recipes on how to solve regularly occurring problems; and guidelines on how to refine, optimize and implement designs.

2.3.2 OSI

Pre-1980, computer communication protocols and systems tended to be dominated by proprietary standards such as IBM's System Network Architecture (SNA) [Cyp78] and DEC's Digital Network Architecture (DNA). Proprietary standards led to closed communities of computers, where only systems from the same manufacturer could interwork. To address this problem the International Organization for Standardization started work, in 1977, on the OSI-RM (Open Systems Interconnection — Reference Model [ISO84]). OSI is an attempt to steer the industry away from proprietary standards and towards open, vendor independent interconnection (interworking) [Lin89].

2.3.2.1 Project history and progress

OSI was originally set up in 1977 by ISO with input from the CCITT (International Telegraph and Telephone Consultative Committee). OSI defines seven communication protocol layers — the commonly known "seven layer model" [Hal88, BS81]. In 1986 ISO, then joined by the IEC (International Electrotechnical Commission), established the Joint Technical Committee 1 Sub-committee 6 (JTC1 SC6) to define the lower protocol layers, and JTC1 SC21 to define the upper upper protocol layers and general architecture. Service and Protocol Standards are now in place for most of the layers.

2.3.2.2 ISO architectural concepts

A number of OSI architectural concepts have been defined. These are used to describe the OSI-RM. [Tur87, TvS92] formally define and categorize the OSI architectural concepts (see section 4.3.3).

2.3.2.3 Formalising OSI

The FDTs Estelle, LOTOS and SDL have been widely used for specifying communications services and protocols. Actual LOTOS specifications of OSI services and protocols include:

- network layer service [Tur89c], network layer protocol [PAN89]
- transport layer service [ISO90a], transport layer protocol [ISO90b]
- session layer service [ISO90c], session layer protocol [ISO90d]
- ROSE (Remote Operations Service Elements) [FA88]
- CCR (Commitment, Concurrency and Recovery) service [Sad90], CCR protocol [JC90]
- TP (Transaction Processing) [WvHR90].

A large body of knowledge on how to specify formally (communications) systems using the FDTs, has evolved from OSI work. This expertise includes [Tur93c, ISO91, VSvSB90, vS90, Tur89b, Tur87, Tur93a].

2.3.2.4 Relevance to thesis

The OSI RM itself is not directly relevant to this thesis. However OSI's infra-structure of architectural concepts has inspired our own infra structure of architectural elements (chapter 4). Particularly pertinent is the work of Turner in [Tur87, Tur91, Tur90, Tur88b] which promotes the importance of, and provides LOTOS semantics for, OSI architectural concepts.

Also, the documented knowledge describing how LOTOS has been used to formalise parts of OSI, has been a guide in our own work of formalising parts of the CIM OSA Reference Architecture using LOTOS (chapter 5).

2.3.3 ODP

The term "open system" originated from OSI, where it was used to qualify systems that were capable of supporting standardized communications protocols. Today openness is a demand made, not only of communications platforms but also, of computing and information platforms. ODP (Open Distributed Processing) [ISO89e, vG89, Lin91, Ste91] is an attempt to meet the new and increasing requirements for openness, and to create a framework for the development of standardized, integrated distributed application platforms.

2.3.3.1 ODP/DAF history and structure

Both ISO and the CCITT have established groups with the objective of producing a Reference Model for ODP. To be exact, creation of a Reference Model for ODP (ODP-RM) is the aim of working group ISO/IEC JTC1/SC21/WG7. CCITT, Question 19, Study Group VII has the task of creating DAF (Distributed Applications Framework) [CCI88a].

Work on the ODP RM began in 1987. The ODP RM is to provide a standardized, conceptual architecture for the design of integrated, distributed environments that span heterogeneous systems. The ODP RM deals with distribution transparency and application portability problems. ODP extends OSI work, using OSI as a means of achieving interconnection and interworking.

By 1988, CCITT had developed two major distributed applications: the X.400 message handling system [CCI88c] and the X.500 directory system [CCI88b]. A drawback of these systems was their informal descriptions. As an attempt to right this weak point, for these and other such systems, CCITT launched a study group whose task it was to create a integrated framework for the support of distributed applications (DAF). DAF was to include modelling guidelines, techniques and tools, as well as generic functions for the support of distributed systems.

Presently, many of the goals for the ODP RM and DAF are being jointly investigated by working groups from both ISO and CCITT. These goals include modelling and specification for ODP and DAF [ISO89d, CCI90c, ISO92b, ISO93a, ISO93b], functions and structures for ODP and DAF [ISO89c, CCI90a], architectural semantics for ODP [ISO93b, ISO93a], DAF security frameworks, ASN-1 and infrastructure. Of these goals, the first three are the most relevant to the work in this thesis.

2.3.3.2 Viewpoints

ODP details five viewpoints of interest. A viewpoint is a particular abstraction of a distributed system, oriented to a particular area of concerns. The purpose of the viewpoints is to provide a framework of abstractions. Partitioning the design space makes the design task easier. Also, the viewpoints represent boundaries for ODP standarisation efforts.

Enterprise viewpoint: (The user requirements viewpoint) This describes the overall objectives and goals of the enterprise in which the system operates. Models from this viewpoint capture the business requirements that mould and govern the design of the distributed system. The enterprise viewpoint is concerned with social, organizational and political issues of an enterprise that affect the distributed system.

Information viewpoint: (The conceptual design and specification viewpoint) This viewpoint concentrates on information structures and information flows within an enterprise. Both manual and automated information processing may be described from this viewpoint.

Computation viewpoint: (The software design viewpoint) The concerns of this viewpoint include the operational and computational parts of the enterprise that modify information. The computation viewpoint describes the organization of, the communications between, and the data structures used by application programs that run on the distributed computing system.

Engineering viewpoint: (The infrastructure of building blocks viewpoint) This describes the engineering structures, mechanisms and tools (such as operating systems, communications protocols, transparency mechanisms, etc.) required to support information processing. The engineering viewpoint addresses concerns such as trade-offs between quality of service attributes such as reliability, portability, performance and maintenance.

Technology viewpoint: (The realised components viewpoint) This viewpoint describes the actual technological artifacts (implemented components) out of which the distributed system is built. Technological artifacts include hardware and software such as operating systems, input/output devices, storage systems, and communication terminals.

2.3.3.3 Structures and functions

ODP aims to develop generic models describing the structures and functions of most distributed systems.

From the computational viewpoint, ODP sees a distributed system as an abstract machine that supports application programs developed using the ODP computational model. The computational model uses objects as units of structure, and interfaces and operations as the basis for interactions.

From an engineering viewpoint, a distributed system consists of nodes (spatially distinct computers), each supporting an ODP nucleus. A nucleus encapsulates computing resources at a node to support the execution of engineering objects. Engineering objects are runtime representations of computational objects. Common engineering objects include transparency mechanisms, activation/passivation mechanisms, failure handlers, migration enablers, etc.

2.3.3.4 Modelling and specification

The ODP work on modelling and specification has two tasks: to identify and define modelling concepts of ODP [ISO89d], and to formally interpret these modelling concepts in FDTs [ISO92b, ISO93a, ISO93b]. The modelling concepts for ODP are divided into three categories: basic modelling concepts, specification concepts and architectural concepts (see section 4.3.1 for details). The work of formally interpreting the modelling concepts uses the FDTs LOTOS, Estelle, SDL and Z. ODP recognizes the importance of providing precise mathematical definitions of its modelling concepts. So far, the work of formalising modelling concepts has been completed for only a limited subset of concepts. [CCI90b] assesses the expressive power of LOTOS with respect to formalising ODP modelling concepts.

2.3.3.5 Relevance to thesis

ODP is relevant to this thesis for several reasons. Chapter 4 uses some of the ODP modelling concepts as a starting basis for our own infra-structure of architecture for distributed systems. We modify these concepts, suggest XL representations for them,

Engineering viewpoint: (The infrastructure of building blocks viewpoint) This describes the engineering structures, mechanisms and tools (such as operating systems, communications protocols, transparency mechanisms, etc.) required to support information processing. The engineering viewpoint addresses concerns such as trade-offs between quality of service attributes such as reliability, portability, performance and maintenance.

Technology viewpoint: (The realised components viewpoint) This viewpoint describes the actual technological artifacts (implemented components) out of which the distributed system is built. Technological artifacts include hardware and software such as operating systems, input/output devices, storage systems, and communication terminals.

2.3.3.3 Structures and functions

ODP aims to develop generic models describing the structures and functions of most distributed systems.

From the computational viewpoint, ODP sees a distributed system as an abstract machine that supports application programs developed using the ODP computational model. The computational model uses objects as units of structure, and interfaces and operations as the basis for interactions.

From an engineering viewpoint, a distributed system consists of nodes (spatially distinct computers), each supporting an ODP nucleus. A nucleus encapsulates computing resources at a node to support the execution of engineering objects. Engineering objects are runtime representations of computational objects. Common engineering objects include transparency mechanisms, activation/passivation mechanisms, failure handlers, migration enablers, etc.

2.3.3.4 Modelling and specification

The ODP work on modelling and specification has two tasks: to identify and define modelling concepts of ODP [ISO89d], and to formally interpret these modelling concepts in FDTs [ISO92b, ISO93a, ISO93b]. The modelling concepts for ODP are divided into three categories: basic modelling concepts, specification concepts and architectural concepts (see section 4.3.1 for details). The work of formally interpreting the modelling concepts uses the FDTs LOTOS, Estelle, SDL and Z. ODP recognizes the importance of providing precise mathematical definitions of its modelling concepts. So far, the work of formalising modelling concepts has been completed for only a limited subset of concepts. [CCI90b] assesses the expressive power of LOTOS with respect to formalising ODP modelling concepts.

2.3.3.5 Relevance to thesis

ODP is relevant to this thesis for several reasons. Chapter 4 uses some of the ODP modelling concepts as a starting basis for our own infra-structure of architecture for distributed systems. We modify these concepts, suggest XL representations for them, and integrate them into our own infra-structure. The ODP work on the formal interpretation of modelling concepts is similar in purpose, although not in form, to chapter 4's efforts to create a unified, (partially) formalized architecture. Also, ODP work on requirements for specification techniques concludes that present FDTs lack "coverage" in areas such as real-time and probability. The extensions to LOTOS described in chapters 6, 7 and 8, address this conclusion.

2.3.4 ANSA/ISA

ANSA/ISA (Advanced Networked Systems Architecture/Integrated Systems Architecture) [ANS89b, ANS89a, ANS86] is a more *applied* architecture than ODP. ANSA is an architecture for building distributed systems which renders the actual distribution transparent to applications, making the distributed system appear as a unified unit. The ANSA ethos is to take full advantage of the parallelism and separation in distributed systems to increase performance and reliability, while hiding disadvantages such as partial failures and communication errors.

2.3.4.1 Project history and activities

Architecture Projects Management Ltd. (APM) was established as a company in 1989 to manage the work originating from the Alvey ANSA project [ANS89a]. ANSA became funded through the Esprit II programme as the ISA (Integrated Systems Architecture) project.

ANSA supports four main activities: developing an architecture for building distributed systems, consisting of design recipes, guidelines, structures and functions; developing software (the "ANSA Testbench") to demonstrate the architecture; contributing to international standards; and technology transfer to the distributed systems community.

2.3.4.2 Architecture

ANSA defines five related models within its architecture. These are called enterprise, information, computation, engineering and technology. These models have extents similar to those of ODP viewpoints (section 2.3.3.2). Of these models, computation and engineering most influence the design of ANSA distributed systems.

The ANSA philosophy is to represent distributed computing concepts, at the programming language level, by additional syntactic constructs. These constructs can then be compiled into system calls. ANSA claims, for this approach, the advantages of a simple programming model, compile-time rather than run-time checking, and independence between the application programmer world and the system world.

The computation model provides a framework for application programming. This model addresses topics such as distributed application modularity, access to services, parameter passing schemes, replication, configuration, concurrency and synchronization. The engineering model provides a framework of compiler and operating system components. These components include: thread and task management, address space management, inter-address space communication, distributed application protocols, interface locators, interface traders, configuration managers, atomic operation managers, and replication managers.

The trader and configuration manager provide means to link applications in a running ANSA system. The trader provides directory information which can be searched in a number of ways. Servers export their interface references to the trader, to make them accessible by other applications. Client applications use the import operation to acquire exported interfaces from the trader. Federation results from linking distributed traders. The job of the configuration manager is to start new applications in a running ANSA system.

2.3.4.3 "Reversed assumptions"

ANSA believes that designers of applications have often taken advantage of a number of simplifying assumptions that are valid only for non-distributed hosts. To right these traditional assumptions, ANSA includes a number of "reversed assumptions" in their architecture and products. These reversed assumptions remain valid in contexts where remoteness would render traditional assumptions invalid. ANSA's approach is to assume that everything is remote, and then engineer local optimization back in where possible. ANSA point out that this is possible because "local failure semantics are always a subset of remote failure semantics". The following list of reverse assumptions appears in [ANS89a], and provides a quick comparison of the problem space of non distributed systems and the problem space of distributed systems.

local	→	remote
direct	→	indirect
sequential	→	concurrent
synchronous	→	asychronous
homogeneous	→	heterogeneous
single instance	→	replicated group
fixed location	→	migrating
shared memory	→	disjoint memories
global name space	→	federated name spaces

2.3.4.4 Transparencies

ANSA offers to the programmer a number of distribution transparencies. The choice of transparencies determines the extent to which programmers must consider, support and control the integration of spatially distributed pieces of the application. In a non transparent program, the application programmer is fully accountable for all aspects of distribution. In a fully transparent program, the application programmer has delegated accountability for distribution problems to the ANSA support environment. ANSA provides support for the following transparencies:

Access transparency provides identical invocation semantics for local and remote components.

Location transparency hides the locations of components so that code does not become dependent on component location.

Concurrency transparency masks the existence of concurrent users of a service such that a client will be unaware of other concurrent clients.

Failure transparency masks the effects of communications errors and partial failures.

Replication transparency disguises any effects of having multiple copies of components (for reliability or performance reasons).

Migration transparency is the dynamic form of location transparency which allows components to be relocated while being used by other components.

Performance transparency allows the system to be reconfigured to improve performance as loads vary.

Scaling transparency allows the system and applications to change in scale without change to the system structure or application program algorithms.

The aim is the support all of these transparencies within ANSA's Testbench.

2.3.4.5 Modelling theory and formal support

ANSA modelling theory [Toc90] and formal support for the development of distributed systems [Toc89], are the two specific areas of ANSA work that are most relevant to this thesis.

ANSA have built a reference dictionary of modelling concepts for distributed systems [ANS89b]. These concepts are defined using a set notation and a graphical syntax [Toc90]. See section 4.3.2 for a list of ANSA modelling concepts.

ANSA believes in formalism to support the design process for distributed systems; formalism ought to support the specification of partial designs at arbitrary levels of abstraction, transformations between designs at the same level of abstraction, refinement of designs to more detailed designs, and separation of concerns. Like ODP, ANSA believes that no one of the existing formal languages meets all of these requirements. However, unlike ODP, ANSA have not opted to use any of the FDT's but instead, have developed their own formalism known as Object Engineering. It is defined using set theory and has a graphical syntax. Objects, interfaces and alphabets are its three principle components. The Object Engineering calculus defines a number of laws of equivalences, refinements, etc. for objects, interfaces and alphabets. [Toc90] describes some of these laws using Z.

2.3.4.6 Relevance to thesis

The ANSA work on programming language support for distributed systems is not directly relevant to this thesis. However, ANSA's modelling concepts and its ideas about formal support and Object Engineering are particularly relevant for chapter 4.

2.3.5 CIM-OSA

The CIM-OSA project (Computer Integrated Manufacturing – Open Systems Architecture [CIM90d, CIM90a, CIM89c, Bee89]) is important to this thesis. Work involving the author [McC91a, MBB90, McC90a, McC90b] to formalise the IIS (Integrating Infrastructure — one part of the CIM-OSA Reference Architecture) has prompted and fuelled the work reported in this thesis. The attempts within CIM-OSA to formalise aspects of CIM systems lead the author to recognize both the weakness of LOTOS for the specification of performance concerns, and the need for an architectural framework to guide specification. To address the first weakness, the author has extended LOTOS for the specification of performance (chapters 6, 7 and 8) and used the CIM-OSA IIS as a case-study (chapter 5) to demonstrate the power of Extended LOTOS. To address the second concern the author has outlined an infra-structure of formalised architectural elements (chapter 4) and has used this as a guide for structuring the specifications in the CIM-OSA case-study of chapter 5.

Due to the importance of CIM-OSA for this thesis, this subsection introduces the history, objectives and structure of the CIM-OSA project, in more depth than for the other reference architectures in this chapter.

2.3.5.1 Introduction

Today's manufacturing enterprises are aware of the strategic importance that complete integration of their manufacturing and business elements within an IT (Information Technology) framework would bring. The ESPRIT II CIM-OSA project defines a reference architecture to guide such enterprise modelling and integration [ESP88, ESP87].

Within this subsection we provide an overview of the CIM-OSA project. We also introduce the IIS — the distributed IT platform on which all CIM-OSA systems are to be built. Later, in chapter 5, we will use the IIS as a case study for the ideas developed in this thesis.

2.3.5.2 CIM in general

To survive, grow and even maintain their position in today's highly competitive market-place, business enterprises have to compete on price, quality and delivery time. To meet such challenges, enterprises employ several strategies, e.g.: optimization of the production process; education of employees; technology management focussing on innovation and internal technology transfer; and integration of the enterprise within an IT framework. CIM (Computer Integrated Manufacturing) mainly encompasses the last of these strategies.

CIM is an area of considerable strategic importance for users, designers/developers, and vendors alike. We can identify three main areas within CIM:

- Design rules, architectures, communications and interfaces — aimed at creating a unified framework conforming to reference models such as OSI, DAF and ODP. This area of research contributes primarily to reductions in the cost of designing, installing and maintaining manufacturing systems.
- Methods and tools for real time manufacturing control. This area of research contributes primarily to reductions in lead times, inventory, unit costs, and to improvements in plant flexibility and productivity to maximize return on investment.

- Shop-floor systems (CIM Technologies). These include robot controllers, sensors for welding, assembly and inspection systems, and various simulation tools. CIM users have committed considerable investment in state-of-the-art Flexible Manufacturing Systems (FMSs) and Flexible Assembly Systems (FASs) [SB87, Hal88, JBD89]. This area of research contributes to improvements in product quality, plant reliability, faster throughput and reduced work in progress.

CIM-OSA is mainly concerned with the first of these three areas.

2.3.5.3 History of the CIM-OSA project

Once the need to develop an Open Systems Architecture for CIM was recognized, Project 688 CIM-OSA was launched by the Commission of the European Communities within its ESPRIT (European Strategic Programme for Research and Development in Information Technology) framework in 1986. The ESPRIT CIM-OSA Consortium, AMICE, consisted of 21 companies from 7 European countries. AMICE represents nearly all of the major European IT vendors, along with major CIM users such as automotive and aerospace industries. Also involved were CIM implementors (e.g. software houses) and university research groups (e.g. University of Stirling). The list of involved companies includes the likes of: Aerospatiale (France); APT (AT&T Netherlands); British Aerospace; Bull; Cap Gemini Sesa (Belgium — who are the prime contractors); Digital; Fiat; GEC; Hewlett Packard; IBM; ICL; Siemens; Volkswagen; etc.

2.3.5.4 CIM-OSA objectives

The primary AMICE objectives are: to create CIM system models and guidelines, to promote industrial cooperation, to influence international standards, to provide bottom-up integration for CIM systems, and to achieve industrial acceptance.

CIM-OSA addresses the problems and needs found in today's manufacturing industries. These problems include:

- The ability to manage change in view of the changing environment.
- The integration of information processing — CIM needs to overcome the problems of fragmented/distributed information processing. Fragmentation arises because of boundaries created due to the use of non-compatible multi-vendor hardware and software, and organizational boundaries within a company. These boundaries lead to inaccessibility and inconsistency of available information.
- Real time control of total manufacturing process, from material input at the supplier to product service at the customer.
- Adaptability and flexibility of the total enterprise (operation and organization).
- An explicitly *processable*, functional and dynamic-behavioural description of the *total* enterprise (for real time operation control and simulation).
- The ability to integrate equipment from different vendors.

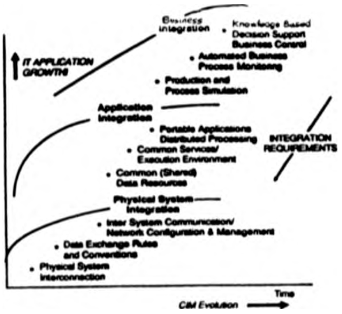

Business Integration
• Knowledge Based Decision Support Business Control
IT APPLICATION GROWTH
• Automated Business Process Monitoring
• Production and Process Simulation
Application Integration
• Portable Applications Distributed Processing
INTEGRATION REQUIREMENTS
• Common Services/ Execution Environment
• Common (Shared) Data Resources
Physical System Integration
• Inter System Communication/ Network Configuration & Management
• Data Exchange Rules and Conventions
• Physical System Interconnection
Time
CIM Evolution

Figure 2.1: Steps towards the integrated enterprise (from [CIM89c])

Figure 2.1 illustrates the steps towards an integrated enterprise. The first level is physical system integration. The physical interconnection of multi-vendor systems is the first problem met on the way towards complete integration. It is essential for attaining higher levels of integration. The need for integration at this level is already being addressed though a number of concepts such as OSI, MMS (Manufacturing Message Specification), etc. The next level, application integration, deals with: information exchange between applications; transportability of applications between different physical systems; distribution of applications; and standardized user interfaces. Limited efforts (such as ODP and DAF) exist for this domain. The highest level of integration is business integration. This deals with the integration of business functions, e.g. design, production, marketing, finance, etc. Note that these functions are concerned with both the running of an operational system and with building/evolving the future system. A CIM system exhibits dynamic behaviour — it is updated according to the evolution of the enterprise business requirements and available technology.

2.3.5.5 Overview of the CIM-OSA Reference Architecture

CIM-OSA provides a CIM Reference Architecture which still allows individual companies to optimize their particular CIM architecture according to their own specific requirements. The CIM Reference Architecture defines generic structures (framework guidelines) that can be used to create a completely structured description of an enterprise as a system.

Figure 2.2 shows the "CIM-OSA Cube" — this is a very general representation of

CIM-OSA's modelling approach.

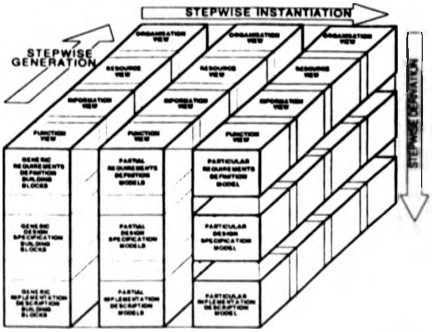

STEPWISE INSTANTIATION
STEPWISE GENERATION
STEPWISE DERIVATION
ORGANISATION VIEW
RESOURCE VIEW
INFORMATION VIEW
FUNCTION VIEW
GENERIC REQUIREMENTS DEFINITION BUILDING BLOCKS
GENERIC DESIGN SPECIFICATION BUILDING BLOCKS
GENERIC IMPLEMENTATION DESCRIPTION BUILDING BLOCKS
PARTIAL REQUIREMENTS DEFINITION MODELS
PARTIAL DESIGN SPECIFICATION MODELS
PARTIAL IMPLEMENTATION DESCRIPTION MODELS
PARTICULAR REQUIREMENTS DEFINITION MODEL
PARTICULAR DESIGN SPECIFICATION MODEL
PARTICULAR IMPLEMENTATION DESCRIPTION MODEL

Figure 2.2: Overview of architectural framework (from [CIM90a])

Below we describe the important aspects of the CIM-OSA framework, and their relations.

Architectural Levels: CIM-OSA Architectural Levels are depicted in figure 2.3. We find Generic Building Blocks at the Generic Level and macro-like constructs called Partial Models at the Partial Level.

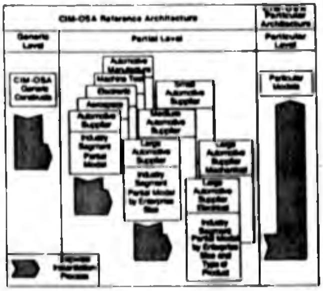

CIM-OSA Reference Architecture
CIM-OSA Particular Architecture
Generic Level
Partial Level
Particular Level
CIM-OSA Generic Constructs
Automotive Manufacture
Machine Tool
Electronic
Aerospace
Automotive Supplier
Industry Segment Partial Model
Small Automotive Supplier
Medium Automotive Supplier
Large Automotive Supplier
Industry Segment Partial Model by Enterprise Size
Large Automotive Supplier Mechanical
Large Automotive Supplier Electrical
Industry Segment Partial Model by Enterprise Size and Type of Product
Particular Models
Stepwise Instantiation Process

Figure 2.3: Architectural levels (from [CIM90a])

Figure 2.3 helps to illustrate the how design ideas are instantiated, in a stepwise fashion, through the three architectural levels. Stepwise instantiation takes the design from the Generic Level through the Partial Level, to eventually instantiate a Particular Model in the Particular Architecture.

Modelling Levels CIM-OSA uses three modelling levels to define, specify and describe the enterprise. Figure 2.4 illustrates the CIM-OSA Modelling Levels and the Derivation Process for a CIM OSA Particular Architecture.

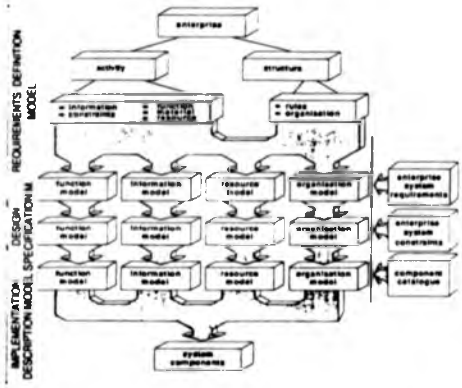


Figure 2.4: Modelling Levels and the Derivation Process for a CIM-OSA Particular Architecture (from [CIM89c])

The Requirements Definition Modelling Level is used to express the system requirements. Here, what needs to be done within the enterprise is described in a business sense using business terminology. This is the domain of the policy makers.

The Design Specification Modelling Level is used to perform system design and model optimization (using computer simulation, etc.). This is the domain of system organizers.

The Implementation Description Modelling Level is used to describe the implementation of the enterprise system. At this level, the integrated set of components necessary for effective realisation of the enterprise operations are implemented. This is the domain of the implementors.

Views Each of the different Modelling Levels and the Architectural Levels are described in according to four different viewpoints. The Function View focuses on the functional structure of the enterprise. The Information View deals with the structure and content of information. The Resource View describes and organizes the enterprise resources. The Organisation View fixes the organizational structure of the enterprise.

2.3.5.6 The IEE and IEO

Figure 2.5 shows the two integrated environments for building and operating a CIM system, that CIM-OSA provides.

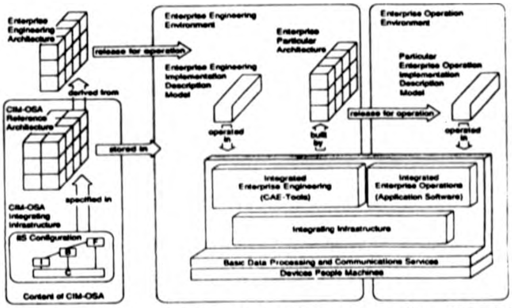

Enterprise Engineering Architecture
release for operation
derived from
CIM-OSA Reference Architecture
stored in
specified in
CIM-OSA Integrating Infrastructure
IIS Configuration
F
B
I
C
Content of CIM-OSA
Enterprise Engineering Environment
Enterprise Engineering Implementation Description Model
operated in
Enterprise Particular Architecture
built by
release for operation
Enterprise Operation Environment
Particular Enterprise Operation Implementation Description Model
operated in
Integrated Enterprise Engineering (CAE-Tools)
Integrated Enterprise Operations (Application Software)
Integrating Infrastructure
Basic Data Processing and Communications Services
Devices People Machines

Figure 2.5: Overview of CIM-OSA integrated environments (from [CIM89c])

The Integrated Enterprise Engineering environment (IEE) covers all build time aspects, including design and maintanance of the CIM system. It comprises Computer-Aided Engineering tools which support the application of CIM OSA's enterprise modelling framework, resulting in a Particular Implementation Model.

The Integrated Enterprise Operations environment (IEO) covers the run-time aspects of the CIM system. It allows the Particular Implementation Model, built in the IEE, to be executed after it has been released for operation.

2.3.5.7 Integrating Infrastructure

Integration at the level of business functions can be reached only if a sufficient level of integration between applications and physical systems is realised (see figure 2.1). In a step towards this objective, CIM-OSA has identified what it considers as a set of services common to most CIM systems. This composite set of services is known as the Integrating Infrastructure (IIS). The IIS acts as an Information Technology platform onto which any Particular CIM-OSA System can be built. The IIS is sometimes known as the "CIM OSA Operating System".

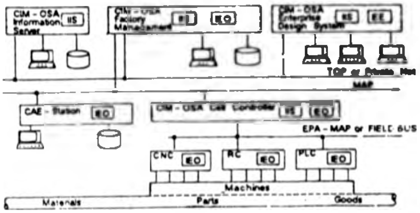

CIM - OSA Information Server
IIS
CIM - OSA Factory Management
EO
CIM - OSA Enterprise Design System
EE
TOP or Private Net
MAP
CAE - Station
EO
CIM - OSA Cell Controller
IIS
EO
EPA - MAP or FIELD BUS
CNC
RC
PLC
Machines
Materials
Parts
Goods

Figure 2.6: Example of integrated environments in use (from [Bee89])

The IIS is a strongly distributed system, as a consequence of the physical distribution found in all CIM systems. The global functionality of the IIS is distributed across several nodes, see figure 2.6. The IIS offers system wide services to CIM applications and users. IIS system wide services are actually realised by a number of physically distributed, inter-working IIS nodes. Section 2.3.5 describes the IIS in detail.

2.3.5.8 The use of formality within CIM-OSA

Once the "Formal Reference Base" (FRB) (which includes [CIM89b, CIM90b, CIM90c, CIM89a]) (the CIM OSA "bible") was established, the need for formal descriptions became apparent. For, despite its name, the FRB consists of (systematic but) informal descriptions (English language text with supporting diagrams). The FRB descriptions were found to be ambiguous and incomplete. CIM-OSA chose to use LOTOS to describe and develop a formal model for the IIS (see section 5.1.3 for details). The attempts within CIM OSA to formalise aspects of CIM-OSA involved the author, and has lead him to recognize both the weakness of LOTOS for the specification of performance concerns, and the need for an architectural framework to guide specification. The work in this thesis finds solutions to these difficulties and applies these solutions (in chapter 5) to a simple CIM-OSA case-study.

2.3.5.9 Results summary from the use of LOTOS for CIM-OSA

In the short time that LOTOS was employed within the project, it had a considerable impact. The use of formalism gained project wide acceptance as a necessary tool for the design of such large and complex systems.

The process of formalization highlighted those aspects of the so called "Formal Reference Base" (consisting of English prose and accompanying diagrams) that were inadequate, incomplete, inconsistent and wrong. Also, the use of formalism — acting as a 'common language' — helped induce coherence between the project workpackages that were working on individual aspects of the IIS.

The general conclusion was that the rigour of formalism promoted early identification of errors and so helped the development of better designs.

2.4 Key issues in distributed systems research for this thesis

This section highlights the key areas within distributed systems research that are addressed in this thesis.

2.4.1 Architecture driven specification

Distributed systems are complex to specify and design. When solution engineers in other disciplines such as civil engineering or electronic design are faced with complex design tasks, they consult their discipline's design guides for *how to* knowledge. Recognising the importance of this paradigm, computing science has begun to establish a number of design guides for various sub fields within its discipline. We have précised a selection of reference architectures (design guides) for distributed systems in section2.3.

Reference architectures support *architecture driven specification*. The notion of architecture driven specification describes a scenario where a specifier is guided in the process of creating a specification by a reference architecture. This guidance takes many forms, including: providing concise terminology and precise concepts for talking about the design space; structuring the design domain by partitioning problems and separating concerns to make the domain easier to understand; predefining generic components that can be customised, or common components that can be re-used; imparting domain knowledge and expertise which have been evolved by previous designers.

We choose architecture driven specification as a key issue in distributed systems research for this thesis. This thesis supports architecture driven specification by creating an infra structure of architectural elements for the formal specification of distributed systems. The features of our infra structure are:

- it is created with specification in mind
- it is hierarchical — founded upon the very general *principles for description*, and rising to more specific *common architectural components*
- Extended LOTOS (XL) representations and process templates are suggested for the architectural elements
- higher level architectural elements are given graphical representations — this makes it much easier to understand 'at a glance' the specification structure
- the categorization of architectural elements provides clues as to how specifications ought to be structured
- XL is used as the formal language because it supports the specification of performance concerns (i.e. timing, probability and priority)

- architectural components are defined for the specification of performance aspects — these performance aspects treated equally with functional and structural aspects
- architectural components may be combined to create either constraint-oriented or resource-oriented specifications.

2.4.2 The specification of performance concerns

The spatial separation of components within a distributed system makes dealing with performance issues very difficult, and elevates these issues to an importance not found in non-distributed systems. Performance issues include: time critical communications, adequate performance, resilience to errors, robust interworking, probability of failures, acceptable delay times, measurement of network dynamics, tolerance metrics, message priority, local clock adjustment, specification of mean failure times, and load requirements.[2] In non-distributed, single host systems these issues either do not arise or can be dealt with easily, and hence play an insignificant role.

The disparity between the perceived importance of performance issues and functionality issues is justified when dealing with non-distributed systems, but the disparity has become obsolete in the move towards distributed systems. A consequence is that specification languages for distributed systems ought to have features for the expression of performance concerns, whereas the specification languages that are used solely for non-distributed systems need not have performance features.

In this thesis we use LOTOS as the basic formal language for specifying distributed systems. LOTOS is good at expressing functionality concerns but poor at expressing performance concerns. The extension of LOTOS for the specification of performance concerns is thus another key area of distributed systems research addressed in this thesis. In chapters 6, 7 and 8 we define quantitative timing, probabilistic and priority extensions to LOTOS; the result being XL — Extended LOTOS. XL is used in chapter 4 to provide formal representations of architectural elements, and in chapter 5 to formalise aspects of the CIMOSA IIS.

2.5 Summary

This chapter introduced distributed systems research as the context of the work in this thesis.

An examination of the theoretical/academic perceptions of distributed systems set the scene. Then we looked at the industrial/application perceptions of distributed systems. Particular attention was paid to the reference architectures OSI, ODP, ANSA and CIM-OSA because of the relevance for chapter 4 of their ideas about architectural concepts, and the consequences for chapters 6, 7 and 8 of their conclusions about the formal specification of distributed systems. CIM-OSA was given a special introduction since

[2]These performance issues are also known as "Quality of Service (QoS)" issues in the specific case of communications and distributed systems.

many of the ideas in this thesis originated from the author's involvement, and also because CIM-OSA is the basis of our case-study in chapter 5.

Chapter 3

An overview of formal languages

This chapter introduces formal languages, specifically LOTOS, as context information for this thesis.

We start by providing a broad classification of the languages that can be used for system specification. Then we focus on the Formal Description Technique LOTOS. (Later chapters will define extensions to LOTOS, and use Extended LOTOS for distributed system specification.) We briefly examine the factors involved in using formal languages for system development, and conclude by highlighting the key issues of LOTOS language research that are addressed by this thesis.

3.1 Introduction

Our interest lies in languages for the formal specification of distributed systems.

This chapter provides general background information about formal specification languages. This is purely a context setting chapter — it does not contrast our work with existing work. Instead, we detail and contrast existing work (in particular, work relating to timing and probabilistic extensions for other process algebras) where appropriate throughout chapters 6, 7 and 8

3.1.1 The attributes of a specification language

An absolute definition of the characteristics of a specification language is not possible. Often we say that, a specification language describes systems in terms of *properties* rather than in terms of *implementation details* — it describes *what* systems are to do, not *how* they should be built. This is a relative definition, and only really makes sense when interpreted in context with the definition of an implementation language.

Below, we discuss the concept of a specification language with respect to a number of attributes.

Abstraction: Generally it is not possible (or desirable) to describe an object in perfect detail. Normally a description will only capture certain aspects, or abstractions of an object. A good description is one which captures, sufficiently precisely, those abstractions that are required for some given purpose. This general heuristic for description languages is also applicable to specification languages. (Specification languages are a particular subset of description languages, that describe *what* systems do, not *how* they do it.)

Distributed systems are complex objects with many facets. A specification of a system will express exactly those facets of the object that precisely characterize the object for some given purpose. To abstract is the process of selectively omitting certain details, resulting in an abstraction.

It is possible to derive a number of different, but complementary, abstractions of the same object. For example, an abstraction may capture what is primarily a physical shape, colour, smell, noise, aesthetic, functional, performance or logical structure concern. No one specification language is suitable for capturing all of these abstractions; and only some of these abstractions are important to any one discipline. For the discipline of distributed systems specification, abstractions of functionality, performance and logical structure are important.

Modularity: The process of abstraction (and its antonym, refinement) moves us *vertically* between different levels of specification. Operating orthogonally to this is the process of modularization (compartmentalization) which moves us *horizontally* between different parts of a specification, at the same level of abstraction. (Modularity may appear in the form of strict, self-contained modules in the sense of Modula-2 [KP85], or alternatively, in the non-strict, overlapping viewpoints sense of ODP (see section 2.3.3.2).) The orthogonality between the notions of abstraction and modularization is conceptual, but well accepted and therefore useful.

A specification language should allow a specification to be organized as a set of modules with well defined interfaces. Modularization ought to be compositional, that is to say, the specification ought to be exactly the result of composing its modules.

For the specification of a distributed system, the modular organization may reflect the physical structure of the system, or alternatively, the logical separation of the functional concerns of the system.

Constructiveness: A constructive specification describes a mechanism (or algorithm) for 'achieving' the properties a system, whereas a non-constructive specification describes the properties of a system without providing any clues as to how these properties might be achieved.

Constructive specifications suggest, if not dictate, mechanisms (or algorithms) to be incorporated in their implementations. Therefore the more constructive a specification, the more it constrains its possible implementations. This is known as over-specification, and is usually frowned upon as specifications ought to say *what* systems are to do, not *how* they are to achieve it. In contrast to this perceived disadvantage of constructiveness is the advantage of the ability to execute (or "animate", or "simulate") constructive specifications. In this way, constructive specifications may be used as early prototypes.

Formality: Formal languages are languages that are wholly defined in terms of axioms and inference rules — usually mathematical axioms and inference rules such as those of logic and set theory. Unlike natural language which has no precise definitions, a formal language enjoys objective interpretation. Individuals must use the unique set of rules that define a formal language in order to interpret a description written the language.

Formality brings with it a number of advantages. Formal languages are:

- precise — unambiguous and exact
- concise — relatively concise syntax, with little "noise" or padding
- consistent — no contradictions
- analysable — amenable to mathematical reasoning.

Also, there are a number of advantages which stem from formality, although not direct results of formality itself. These include:

- correctness — provable with respect to the requirements
- completeness — no unwanted omissions
- implementability — often formal rules can be given operational interpretations.

Often, the major disadvantage of formalism is:

- incomprehensibility — this is a direct result of the conciseness and unfamiliarity of the formal notation to those without special training.

We would like a language for the specification of distributed systems to have all of the above advantages.

A formal language is not, by itself, a solution to all of the traditional problems associated with the development of distributed systems. However, its qualities do go some way to alleviating the difficulties, and a formal notation together with formal reasoning may result in the automation of parts of the development process.

Concurrency: Section 2.2 explained how concurrency arises in a distributed system as a result of explicit parallelism or the inability to totally order events due to the spatial separation. Concurrency is an important property of distributed systems. Therefore it is essential that a language for the specification of distributed systems, can express concurrency in a direct way and can support reasoning about concurrency issues.

In the main, formal languages adopt one of two formal models of concurrency. The "interleaving" model [Mil80, Hoa83] represents concurrent events by saying that "concurrent" events occur arbitrarily closely in time in arbitrary order. Events within a "permutation sequence" are ordered in time, but the intervals between the events can be arbitrarily small. A trace through an interleaved model includes exactly one of the permutation sequences, chosen arbitrarily. The "true concurrency" model [CdC91] represents concurrent events by a bag of events, all of which are deemed to have occurred at the same instant. A trace through a true concurrency model is a (partially) ordered list of bags of concurrent events. The choice between these two models of concurrency is normally made on philosophical grounds. Pragmatically, there is little difference between the models.

Determinism: A system is deterministic if we can accurately predict its future behaviour. Prediction of future behaviour requires a complete and accurate description of the system's present state and future specification together with a description of the future influences from its environment. (Of course, we cannot accurately predict the future behaviours of real-life systems because of the impossibility of capturing and analysing such information.)

In the world of systems specification a system is non deterministic if, knowing environmental influences, we cannot accurately predict its reactions. A non-deterministic system displays behaviour which is not solely determined by environmental interaction.

Within specifications, non determinism may be used to to describe a set of possible implementations. For instance, a non deterministic choice between two possible behaviours may be interpreted as a choice between two possible implementations.

Description of structure, data and behaviour: Usually, to understand a system we subdivide it into static aspects (structure and data) and dynamic aspects (behaviour). It follows that, for a specification language to be comprehensible, it ought to provide features which allow this natural subdivision to be reflected within its own descriptions.

Quantitative issues: In section 2.4.2 we provided examples of performance concerns, and discussed reasons for their particular importance for distributed systems. Quantitative issues are intrinsic to performance concerns. For the specification of performance concerns, a specification language ought to support suitable quantitative metrics and measures (e.g. for time, probability and priority). Metrics and their supporting mechanisms will preferably be built into the specification language, thus unburdening the user from having to describe these at a higher level.

Reasoning support: Specification languages ought to be supported by a knowledge base of theories for reasoning about specifications written in that language, and supported by tutorial material which explains how to use the language. Formal theories for reasoning about specifications usually include equivalences, implementation and transformation relations. Tutorial material provides examples and guidelines for using the language, and suggests how the user might informally relate the specification language concepts to concepts within his/her understanding.

3.2 A plethora of specification languages

Specification languages may be classified according to their abilities to concisely express certain properties. For example, state oriented languages are good at concisely expressing properties such as 'in state x, y can happen causing...'; transition oriented languages are good at concisely expressing properties such as 'x can happen followed by y...'; and property oriented languages are good at concisely expressing properties such as 'x holds under these circumstances...'. In this section, we use such criteria to create a broad classification of specification languages (see figure 3.1). This classification is not perfect (some languages could be placed in alternative classes), nor is it the only classification possible (we might have used Chomsky's criteria).

3.2.1 Informal specification languages

Informal languages are the the most widely used class of specification languages. An informal specification language is not defined in terms of mathematics and thus is open to misinterpretation.

Specifications written in everyday language usually consist of natural language prose with accompanying diagrams and tables. In order to reduce the scope for misinterpretation, everyday language specifications are often written in "stylised natural language" (see any legal document), or are hierarchically structured with an extensive use of cross-reference labels (see any non-formal ISO document).

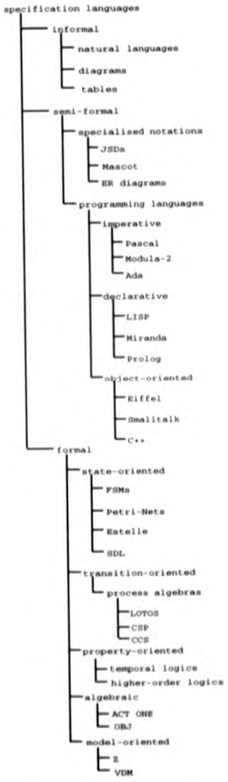


Figure 3.1: A broad classification of specification languages with some examples

3.2.2 Semi-formal specification languages

For a semi-formal specification language, the scope for misinterpretation is reduced because it has a partially complete, agreed definition.

3.2.2.1 Specialised notations

Specialised notations include: Jackson Structured Diagrams [Jac83]; Mascot [Mas80], system flow charts, entity-relationship diagrams and structure diagrams [SMC74, YC79, You89].

3.2.2.2 Programming languages

Programming languages can be used to express both specifications and implementations.

Imperative programming languages Many imperative programming language concepts are closely allied to notions of computer architecture, such as memory and assignment. Because of this, imperative languages tend to be implementation oriented. Also, since traditional computer architectures support what is essentially sequential execution, most imperative languages do not support features for expressing concurrency. Nevertheless, imperative programming languages have been used as specification and design languages, e.g. Modula-2 [KP85], Pascal [ISO82] and Ada [Boo87]). Estelle [ISO89a] is an example of a formal specification language which is based on the programming language Pascal.

Declarative programming languages Declarative programming languages are based on set theory, lambda calculus and logic. Often, they have formal semantics (separate from their computer based realizations), in which case they ought to be classified as formal languages. Declarative languages are abstract in that their concepts are unrelated to computer architecture. Declarative languages allow concurrency. In *functional* declarative languages (such as, *me too* [AJ89] and Miranda [Tur85]) function arguments may be evaluated in parallel. In *logic* declarative languages (such as Prolog [Ham89]) goals-to-prove may be pursued in parallel.

Object-oriented programming languages The basic concept in object-oriented programming languages is the *object*. An object is an automonous entity, encapsulating both data and processing services. A system is built as a set of interacting, possibly concurrent objects. Objects interact by sending and receiving *messages*. Receipt of a message by an object invokes a *method* (a processing service). The method may modify the state of the object, and may send messages to other objects to solicit help, or return a response. In object oriented languages, objects which share a common set of methods and data form a *class*. Each object is an *instance* of a class (an instantiation of a template). Classes are structured in a hierarchical fashion. A class automatically *inherits* generic methods from the class which is directly above it in the class hierarchy. Inheritance is a form of relegation of description.

The object oriented model is well aligned with distributed computing system frameworks such as ODP and ANSA/ISA (see sections 2.3.3 and 2.3.4), in which systems are described in terms of distributed collections of objects relying on underlying communications services (such as OSI) for message-passing.

Examples of object-oriented languages include Eiffel [Mey87, Mey88a, Mey88b], Smalltalk [GR83] and C++ [Str86]. Of these languages, Eiffel in particular is useful for specification, design and implementation stages. This is, in part, because Eiffel allows all stages to be carried out within the same methodology and environment. Another reason for Eiffel's suitability as a specification language is that the Eiffel language includes *assertion* constructs that promote formal reasoning about statements written in the language. The assertion constructs are generally boolean expressions that express some property which should be satisfied by certain entities at a designated stage in the execution of a system. These assertions include: preconditions, postconditions, class invariants and exceptions (explicit assertion violations). Eiffel does not yet have a full assertion language for expressing Eiffel designs in a formal way, but work towards this goal is in progress [Mey93].

3.2.3 Formal specification languages

The attributes of formal specification languages were introduced in section 3.1.1 — formal specification languages have mathematical definitions. A variety of formal specification languages exists. Some of these are outlined and classified below.

3.2.3.1 State-oriented

State oriented and transition-oriented specification languages describe systems in terms of graphs. Graph nodes represent system states and graph arcs represent system transitions. A state-oriented specification language emphasizes graph nodes (states), while a transition-oriented language emphasizes graph arcs (sequences of transitions).

Finite State Machines A finite state machine (FSM) (or finite state automaton, FSA) is an abstract machine with a finite number of states. The machine can be in only one state at a time. The machine starts from one state, known as the initial state. A transition takes the machine from one state to another state. In one kind of FSM, the Mealy Moore machine [HU79], transitions are associated with output or input symbols — input symbols drive the choices between transitions; or output symbols indicate the transitions taken.

The size of a FSM description very quickly becomes unmanageable as the complexity of the system being described increases. This is because, in their basic form, FSMs do not have concepts such as definition and instantiation, variables, parameterization, etc. FSMs that support some of these concepts are known as extended finite state machines. Also, without support for modular composition, FSMs are monolithic in structure.

Another problem with FSMs is that their sequential behaviour might be construed as a requirement for serialized behaviour in an implementation, rather than as an artifact of their model.

Estelle [ISO89a] is an example of a language which is based on FSMs. In sections 6.2.3 and 8.2.3 we use an FSM to explain (Extended) LOTOS specifications.

Petri-Nets Petri-Nets were first introduced by C.A. Petri [Pet62, Pet81]. A Petri-Net is a particular kind of directed graph with two types of nodes, namely the *places* and *transitions*. The basic structure of a Petri-Net consists of a set of places (depicted as circles), a set of transitions (depicted as bars), and a set of directed arcs which connect the transitions and the places. An arc directed from a place to a transition defines that place to be an *input place* of that transition. An arc connecting a transition to a place implies that the place is an *output place* of the transition. A transition may have multiple input places, and multiple output places. The *state* of a Petri-Net is described by the distribution of markers, called *tokens*, in the places of the net. Tokens are depicted as dots inside the places that have them. A particular assignment of tokens is referred to as the *marking* of the net.

A transition can *fire* when each of its input places contains a token. A firing results in one token being deleted from each input place, and one token being added to each output place. The most interesting feature of Petri-Nets is that transitions can fire simultaneously and therefore Petri-Nets model true concurrency.

A large body of theories, techniques and tools exists for Petri-Nets. In fact, the availability of powerful analytical techniques is one of the major advantages of Petri-Nets. For example, there are theories for reachability analysis, invariant analysis, assertion proving, and optimizations.

Various extensions (high level Petri-Nets) exist beyond the basic model. These include Arc-timed Petri Nets, Coloured Petri-Nets, Predicate-Transition Petri-Nets and Numerical Petri Nets [Wal93, Dia91]. They effect compact representations of systems by having each place represent a number of different conditions.

In section 6.3 we use a derivative of Arc-timed Petri-Nets as a vehicle for the exploration of quantitative time concerned specification.

Estelle Estelle (Extended Finite State Machine Language) [ISO89a] is a FDT (Formal Description Language) which has been standardized by ISO and accepted by CCITT (see section 3.3.1). Estelle is based on an extended finite state machine (EFSM) model. An Estelle description consists of a number of hierarchical, communicating, non-deterministic EFSMs. The EFSMs communicate messages via bi-directional *channels* that are connected to *interaction* points. Channels queue messages at either end until an EFSM is ready to accept them. EFSM transitions are described in a derivative of Pascal.

Estelle allows *delay clauses* for expressing timing concerns, *spontaneous transitions*, non-deterministic choice, dynamic creation of EFSMs, and concurrency controls between EFSMs. A Pascal derivative is used to define the data typing part of Estelle.

SDL SDL (Specification and Description Language) [CCI92] is an FDT which has been standardized by CCITT and accepted by ISO. Like Estelle, SDL is based on an EFSM model. Like LOTOS, SDL uses abstract data types (see section 3.2.3.4) for data typing.

An SDL specification is built from a hierarchy of *blocks*. Blocks decompose into sub-blocks and, eventually, *processes*. Processes are represented by EFSMs. *Signals* (messages) travel between blocks via *channels* (bi-directional queues with inspection and priority facilities), and between processes via *signal routes*.

SDL was developed specifically for telecommunication systems specification by CCITT, and has a formal semantics. SDL has two representations: a graphical representation (SDL/GR), and a textual (program like) representation (SDL/PR). SDL is an abstract language, with a mature body of experience and tools.

3.2.3.2 Transition-oriented

Transition-oriented languages characterize systems in terms of sequences of transitions (in contrast to state-oriented languages that characterize systems in terms of states).

Process algebras Process algebras describe the externally observable behaviour of systems in terms of *events* (transitions). Events are often considered to be atomic; an event occurs at a single point in time. Events are used to represent real-world actions. Generally, an event is the result of a *synchronization* between two or more *event offers*. An event offer is associated with a *process* — a structural unit. A system is structured in terms of hierarchies of synchronizing processes. Each process defines ordering constraints over the event offers that are sited within it. The synchronous combination (using a variety of operators) of all processes yields the overall behaviour of the system. Extenally observable events are events which require the synchronous participation of the *environment* (considered to be a process outside the system itself). Events which do not involve the environment are known as *internal events*. These are not observable from outside the system, and are used to model "spontaneous" actions within the system.

Process algebras are particularly apt for the concise specification of concurrency and other temporal ordering concerns. Examples of process algebras include CCS (Calculus of Communicating Systems, [Mil80]) and CSP (Communicating Sequential Processes, [Hoa83, Hoa85]). The process algebra part of LOTOS is largely based on these two algebras.

3.2.3.3 Property-oriented

A distributed system specification may be expressed as a set of properties that the system is required to satisfy. Properties can be formally expressed as *predicates* — boolean functions over sets of mathematical objects. A system satisfies a property if the system representation in terms of mathematical objects satisfies the analogous predicate. Property oriented specifications tend to be very concise, but can be difficult to understand without mathematical training.

Property-oriented languages are based on classic mathematical logic (propositional and predicate calculi, and set theory). Property-oriented languages include *temporal logics* (a form of *modal logic*) and *higher-order logics*.

Temporal logics Temporal logics (or, more generally, *modal logics*) are specialized logics for expressing time-related properties. Classical logics are useful for the formal specification and verification of deterministic sequential systems. However, when distributed system characteristics such as non-determinism and concurrency are involved, the extra expressiveness of temporal logic may be needed to concisely state system properties. Pnueli [Pnu77, Pnu81] proposed using temporal logic for reasoning about such systems, as this has temporal operators with meanings such as *"always"*, *"eventually"* and *"next"* which hide the explicit quantification over time otherwise needed. Temporal logics have been developed into forms useful for reasoning about computing systems by [Pnu77, Pnu81, Lam80, Lam80, Lam77, Lam83, BAPM83].

Linear temporal logics consider time as a sequence of discrete states so that at each moment there is only one possible future, whereas *branching temporal logics* consider time as a tree structure so that at each moment time can branch into alternative futures. Lamport [Lam80] concludes that linear temporal logic is suitable for reasoning about concurrent systems, while branching temporal is preferable for reasoning about non-deterministic systems. However, a study by [EH86] showed that branching temporal logics are needed for reasoning about "existential" properties of concurrent systems such as the possibility of deadlock in some future.

Temporal logics are especially concise for expressing *safety* (invariance or "something bad never happens" [Lam80]), *liveness* (eventuality or "something good must eventually happen"[Lam80]), *precedence* [MP81] and *fairness* [GPSS80].

Specific work on the use of temporal logics for system specification can be found in [BK84, BKP84, Bar87, BKP85, HO83, HdR89]. These references develop compositional temporal logics for modular specification. [FGL90, BKP85] report on work aimed at providing temporal logic semantics for LOTOS, and work towards verifying properties, expressed in temporal logic, of LOTOS systems.

3.2.3.4 Algebraic languages

Algebraic languages concentrate on the logical properties of data types and operations. Algebraic languages do not have facilities for expressing temporal ordering concerns. Algebraic languages abstract from concrete representations of data types and operations by defining only those essential properties of the data and operations that any correct implementation must satisfy. This is achieved by identifying the mathematical object, namely an algebra, which is formed by the sets of data values, called the data carriers, and the sets of operations that can be performed on these. (See [EM85] for a full explanation of algebras.) Examples of algebraic languages include ACT ONE [EM85] and OBJ [OF88, OIF91].

ACT ONE The ACT ONE language was developed within the "Algebraic specification techniques for the correct design of trustworthy software systems" project by the ACT-group at the Technical University of Berlin in 1983. A revised version was later presented in [EM85]. ACT ONE forms the basis of the data typing parts of both LOTOS and SDL.

The notion of *parameterised specification* drove the design of the ACT ONE language.

A parameterised specification is a specification with *formal parameters* which can later be *instantiated* with actual values. ACT ONE provides four concepts for defining and structuring a system: basic specification, *renaming*, *enrichment* and *parameterization/actualization*.

Basic specification involves defining *types*. Types encapsulate sets of values called *sorts* and the *operations* on these. Sorts are defined solely in terms of the operations allowed on them. Operations are defined by means of *equations* which state equivalence classes among *terms* (the expressions formed from operations). The equations behave like *rewrite rules*. Rewrite rules provide an operational interpretation for ACT ONE specifications — if the equations are confluent and terminating (i.e. are Church-Rosser), then repeated application of the rewrite rules will reduce a term to its unique *normal form*.

Renaming makes a copy of a type and changes the sort or operation names of the copy. Enrichment allows types to import and use other types within their own definitions. In this way, a type may extend another type. Actualization instantiates parameterized types with actual values.

3.2.3.5 Model-oriented languages

Model-oriented languages use notations which are specializations of logic and set theory. Using these notations, system behaviours are described in terms of *pre-conditions* and *post-conditions*. Examples of model oriented languages are Z [Spi89] and VDM [BJ78].

3.3 LOTOS

We have chosen to use LOTOS (Language of Temporal Ordering Specification [ISO89b]) as a basis for distributed system specification. Later chapters will define extensions to LOTOS and demonstrate how these extensions may be used to specify key aspects of distributed systems (as discussed in section 2.4.2). This section provides a brief overview of the evolution, features and applications of LOTOS.

3.3.1 The FDTs: the genesis of LOTOS

In 1980, ISO gave a remit to ISO Committee SC21/WG1 to standardize formal description techniques for OSI. Although a number of formal languages already existed, their definitions were prone to uncoordinated changes — none of the existing languages enjoyed management by international standards bodies. Two languages emerged from the work of the SC21/WG1: Estelle, based on FSMs; and LOTOS, based on process algebras.

In parallel with ISO, CCITT had developed their own standarised formal language, SDL, for the specification and design of telecommunications systems. ISO and CCITT agreed to recognise Estelle, LOTOS and SDL as mutually acceptable standards. They became known as the FDTs. (Nowadays, the term FDT is sometimes used less accurately to indicate any formal specification language.)

3.3.2 Features of the LOTOS language

LOTOS was developed by ISO/TC97/SC21/WG1/FDT/Subgroup C to become an International Standard (ISO 8807) in 1988. LOTOS has been designed for the formal specification of distributed, concurrent, information processing systems.

LOTOS adheres to the principles of good language design: it has firm mathematical foundations, and is a small language with powerful constructs. Its mathematical basis supports formal reasoning, and its powerful constructs gives it a rich expressiveness. An introductory tutorial to the language can be found in [ISO89b, BB87], with guidelines for its use in [Tur93c, ISO91, VSvS88, VSvSB90, vS90, Tur89b, Tur89a].

LOTOS is a combination of two distinct formalisms: a process algebra language and an abstract data typing language. The process language is based primarily on Milner's CCS [Mil80] with influences from related calculi, especially Hoare's CSP [Hoa83, Hoa85]. The abstract data typing algebra part is based on ACT ONE [EM85] (section 3.2.3.4).

The basic unit of behavioural structure in a LOTOS specification is the *process*. Processes are treated as "black boxes" LOTOS describes only the externally observable behaviour of processes. The behaviour of a LOTOS specification consists of all the (observable) interactions between the processes and the specification environment. LOTOS models interactions as discrete, atomic *events* (or *actions*).

Events occur at specific interaction points called *gates*. Processes are able to communicate with each other if they share a common set of gates. An event represents both a communication and a synchronization. An event occurs when all the participating processes (including the specification environment, if the event is observable) synchronize. Hence, LOTOS is labelled as a "synchronous algebra". Unlike CCS, LOTOS supports multi-way synchronization between more than two processes.

Only observable events are visible. *Internal events* are represented by special, unobservable **i** events. An internal event occurs spontaneouly without the participation or control of the environment.

A process may be internally structured as a collection of sub-processes. In this way, an entire LOTOS specification is a single process (internally, hierarchically structured into sub-processes) interacting with the environment (which, itself, can be considered to be a special process outside the scope of the specification). Complex behaviour is built by combining simpler components. A LOTOS behaviour is denoted by a *behaviour expression*. The LOTOS language consists of fundamental behaviour expressions and a small, but powerful, group of operators to combine behaviour expressions according to defined rules. These operators can be used for specifying sequencing, choice, nondeterminism, concurrency, synchronized and interleaved behaviour. LOTOS models concurrency using the interleaving model (section 3.1.1).

The static structure of a LOTOS specification is governed by the formal syntax rules of LOTOS. These are given in extended BNF notation.

The formal semantic rules describe the interpretation of a LOTOS specification. The semantics of LOTOS are described in the SOS (Structured Operational Semantics) style of [Plo81]. Each behaviour expression is semantically defined as a labelled transition system. The transition system is constructed by axioms for the fundamental behaviour expressions, and inference schemas for the operators. Each inference schema defines how (at the semantic level) a transition system is derived from two simpler transition systems, in reflection of the way in which (at the syntactic level) a particular operator combines two behaviour expressions.

LOTOS uses a language derived from ACT ONE to represent values, value expressions, data structures, etc. The data typing language has pre-defined library types, *conditional equations*, *renaming*, *enrichment*, *parameterization* and *actualization* features (see section 3.2.3.4). The fusion of the data typing language with the process algebra language allows:

- guarded events — a boolean value expression may guard/condition the occurrence of an event
- value passing — values may be passed from a terminating process to its successor process
- value negotiation — a set of values may be negotiated on event synchronization. Each participating process provides an *event-offer* — a gate, event values and a predicate over event value parameters. When event synchronization occurs, a set of values is negotiated between the participating processes such that the values in the negotiated set satisfy all the predicates associated with the event offers.

The net effect of these language features is to make LOTOS particularly good for capturing abstract descriptions of distributed, concurrent, non-deterministic systems. However, we believe that LOTOS does have a few inadequacies when it comes to the specification of certain quantitative aspects of distributed systems. We have already mentioned this concern in a general discussion in section 2.4.2. Section 3.5 will list the key inadequancies of LOTOS.

3.3.3 Work related to LOTOS

Already, a large body of literature exists which documents work related to LOTOS. Below, we provide references to some of this work. Elsewhere in this thesis, references to additional LOTOS work and other related work are given where appropriate.

Definitive literature: [ISO89b]

Tutorial literature: [BB87, Tur93c, Bri88a, vEVD89, ISO91, VSvSB90, vS90, Tur93b, Tur89a]

LOTOS relations: [Bri88b, Wez89, BS86, Led91a, WBL90, FGM90, Led90]

Specifications of international standards in LOTOS: [ISO92a, ISO90a, ISO90b, ISO90c, ISO90d, Sad90, JC90, WvHR90]

LOTOS method and tool development projects: [Tur89b, QPF89, Mar89, Tur88b, Pir91, BNO88, BWN$^+$88, WB91, Win92, MC93]

LOTOS and architecture: [Tur87, Tur91, VSvS88, Bog90, VSvSB90, TS93, Tur93a, TvS92]

The application of LOTOS to CIM-OSA: [Vio90, BV89, McC90a, McC90b, MBB90]

Timing extensions: [QAF90, vHTZ90, BL91, McC91b, Led91b, Fid90, ERP90]

Object-oriented LOTOS: [vH89, CRS89, Bla89, May89, CJ92, Cla90]

3.4 System development methods using LOTOS

A **system development method** is a particular way of producing an implementation that satisfies a given set of informal requirements. A **formal system development method** (or "**formal method**" for short) is a system development method employing formalism. The term *formal method* is somewhat misleading. All system development methods require informal input and so no method can claim to be completely formal (see [Bri91]), though the method of processing informal requirements can be formal. Realistic goals for formal methods are: partial automation of the development process; and providing means of handling complexity through tool and reasoning support.

A formal method requires a number of ingredients, including:

1. a set of formal languages or notations in which to model the target system. In this thesis, we develop and use Extended LOTOS as our formal language.
2. a collection of formal reasoning rules for guiding and controlling step wise refinements and transformations. Section 3.3.3 provides references to formal relations, transformations and reasoning rules for LOTOS. Appendix G and section 7.5 add to these existing reasoning rules by developing formal testing ideas to support our LOTOS extensions.
3. a reference-base of domain-specific information for providing a design framework. Sections 2.3 and 3.3 cite reference material which suggests how to use LOTOS in the domains of ODP, CIM, OSI and object-oriented systems. Chapters 4 and 5 develop a specification framework for the domain of distributed (performance concerned) systems.
4. tools to automate parts of the development process. We overview tools for supporting LOTOS development methods in section 3.4.3.
5. creativity to originate ideas.

3.4.1 Development activities

[Tur89b] provides a list of the identifiable activities within a formal development method (also see [Tur93c, Chapter 11]); these activities are:

- **Requirements capture** — requirements input cannot be formalised and will occur, in gradually lesser degrees, throughout the development lifetime. Typically, requirements capture involves extracting and assembling in a semi-structured way, a mass of requirements from customer personnel and documents.

- **Formal specification** of requirements — the captured requirements are given a formal representation.
- **Analysing the requirements** through the formal specification — the process of writing the formal specification will highlight ambiguities and inconsistencies in the requirements. The formal specification may be analysed for safety, liveness, freedom from deadlock, etc. properties, and checked for completeness.
- **Design steps** — these progressively move the development focus from the specification level towards the implementation level. Each design step contains two activities:
  - **Refinement of the design** — introduces implementation-oriented detail, solves design problems and resolves design choices. Formal correctness preserving, (semi-)automated transformations may be used (e.g. [Pir91]), or informal guidelines may be used (see section 5.5).
  - **Verification** — proving that the refinement satisfies the specification. Proofs may be based on formal implementation relations and equivalences (e.g. [Bri88b, Wez89, BS86, Led91a], and see appendix G).
- **Implementation** of the design — code generating tools may partially automate this activity (e.g. [MdM89, WB89]).
- **Validation** — testing that the implementation satisfies the specification. Testing frameworks may be based on formal theory and formal relations (e.g. section 7.5, [Bri88b]).

From the inherent ordering in the above list of activities, we might infer that formal development methods take us towards a solution in well-defined successive steps. However, in the next subsection we caution against emphasising such simple development life-cycle models.

3.4.2 Life-cycle models for formal development methods

Development life cycle models, such as the waterfall model or spiral model, are "ivory tower" models. Such models are valuable abstractions and simplifications but, unfortunately these abstract models are sometimes adopted by the formal methods community as realistic models to be used in the industrial world. The structured ness of these models may seem appealing, but it is also unnatural — it does not accommodate the manner in which humans work.

Research results concerned with the systematization of the development process have matured and become less naive over the last 25 years. [CM91] describes how the strict clustering and decompositional structuring techniques of the 1960s (the so-called "classic analysis and design techniques") fail when applied to most real world problems. Empirical studies [VH90, Vis90, Gui90, CM91, LL91, MTCM80] have concluded that humans use both *step-by-step* development (when advancement towards a solution proceeds in successive steps) and *opportunistic* development (when advancement towards a solution proceeds in a seemingly *ad-hoc* fashion).

In opportunistic development, the design space is explored in an apparently non-systematic way. Opportunistic development can best be defined by listing some example characteristics of it in action, e.g.:

- the designer's focus jumps between different levels of abstraction and refinement throughout the development life cycle (although the overall trend is to move from higher to lower levels of abstraction as the life cycle progresses)
- elements from previous (partial) solutions are frequently incorporated into new solutions
- inspiration at one point in the design space frequently de-mystifies other areas of the design space
- partial completion of subgoals, and then backtracking to work on other goals
- partial and incorrect derivations may exist at intermediate stages in the design process
- interleaving work on distinct areas of the design space.

Methods and computer aided software engineering tools based strictly upon the Classic development models (such as top-down, breadth first), limit the opportunities for designers to exercise insight. Such tools restrict the designer from jumping around the design space to opportunistically fill-in multi-dimensional jigsaw-like pieces of the design.

The idea of the opportunistic development life-cycle contrasts with the very systematized life cycle models (such as the waterfall and spiral models) that are often referenced and reworked by the formal methods community. However, the apparent contention between (the systematization from) formal methods and (the *ad-hoc-ness* from) opportunistic development is reconcilable. We believe that a formal method can be supported within a framework which allows for opportunistic development. Development methods and tools could allow designs to be developed in an opportunistic way and yet these designs could be amenable to formal manipulation. Accommodating within formal methods, the *ad-hoc* manner in which humans work, may lead to a wider acceptance of formal methods.

3.4.3 Tools

The effective use of formal specification languages and methods requires the support of software tools. We use [Tur89b]'s broad classification of tools to structure our overview of the tools available to support LOTOS.

3.4.3.1 "Book-keeping tools"

Book-keeping tools are used to create, edit, maintain and print specifications. Normally, operating system utilities, such as text editors and version control systems, are adequate (unless the formal language uses a specialized graphical notation).

3.4.3.2 "Front-end tools"

Front-end tools directly manipulate the specification text. LOTOS front-end tools include [Mar89]:

- syntax directed editors, e.g. a Cornell Synthesizer Generator editor [vE89], graphical editor for G-LOTOS [CYYW89], MELO (Mentor LOTOS Editor), SEAL (Structure Editor Adopted to LOTOS)
- syntax checkers, e.g. SCLOTOS (Syntax Checker for LOTOS)
- cross-referencers, e.g. LXREF (LOTOS Cross Referencer)
- abstract representation builders, e.g. LASTB (LOTOS Abstract Syntax Tree Builder)
- static semantics analysers, e.g. LISA (LOTOS Integrated Static Analyser)
- print formatters, e.g. PPLOTOS (Pretty Printer for LOTOS).

3.4.3.3 Analysis tools

Analysis tools are used to verify, extract interpretation from, and symbolically execute formal specifications. LOTOS analysis tools include:

- AUTO [MV89] — a tool for the analysis and manipulation of labelled transition systems
- SQUIGGLES [BC89]: (named after the symbols $\approx$ and $\sim$ denoting strong and weak observational equivalences) — a tool for automatically checking the strong bisimulation of LOTOS specifications
- PERLON [BdMS89] — tool for analysing the persistency properties of ACT ONE data types
- symbolic executors (animators, simulators, interpreters), e.g. SMILE [vEE91], the Ottawa interpreter [LOBF88], HIPPO [vE88, Mar89], SPIDER (Service and Protocol Interactive Development Environment) [Joh89].

3.4.3.4 "Back-end tools"

Back-end tools are used to transform and implement specifications. LOTOS back end tools include:

- LOLA (LOTOS Laboratory) [QPF89] — a tool for the automatic expansion of LOTOS specifications
- COOPER [Ald90] — a tool for the derivation of canonical testers based on the COOP method [Wez89]
- TOPO [MdM89] — generates C code from LOTOS specifications

- compilation of ACT ONE into abstract term rewriting machines is reported in [WB89].

3.5 Key issues in formal language research for this thesis

This section highlights the key areas within formal language research that are addressed in this thesis.

The thesis delves into three broad areas of research:

- enhancing the LOTOS language
- architectural concepts and their definition
- application of the LOTOS language.

3.5.1 Research into LOTOS language enhancements

A number of enhancements to the LOTOS language have already been proposed (e.g. [ISO93c, Pir91, QAF90, vHTZ90, BL91, Led91b]). Some of these concur with our own requirements for enhancing LOTOS, some do not. Where appropriate (particularly in sections 6.4 and 7.2) we reference existing proposals for LOTOS language enhancements, and compare these proposals with our own proposals.

Our concern in this thesis lies with using LOTOS as a basis for the formal specification of distributed systems. Section 3.1.1 details the attributes required of a formal specification language for distributed systems. LOTOS scores favourably on all but one count — the expression of *quantitative issues*.

The ability to express quantitative information is necessary for the specification of performance (or Quality of Service) concerns for distributed systems (section 2.4.2 discusses the importance of this). To remedy this inadequacy of the LOTOS language, we propose and define three extensions to the language:

TLOTOS: LOTOS extended for the formal specification of quantitative timing concerns — chapter 6.

PbLOTOS: LOTOS extended for the formal specification of probabilistic concerns — chapter 7.

PrLOTOS: LOTOS extended for the formal specification of priority concerns — section 8.1.

We define the enhancements to the LOTOS syntax and semantics required to realize each of these extensions, and provide examples for the use of each extension. Also, for TLOTOS and PbLOTOS we define formal equivalences and implementation relations, and discuss testing. These extensions may be used in isolation or in combination with one another. We call the combination of all three extensions (TLOTOS+PbLOTOS+PrLOTOS) **Extended LOTOS (XL)**. Section 8.2 describes the integration of TLOTOS, PbLOTOS and PrLOTOS, and provides examples in the use of XL.

3.5.2 Research into architectural concepts

The thesis develops a 'method' for the application of (Extended) LOTOS, called architecture-driven specification (chapter 4). Our architectural concepts provide both *functional components* and *performance components* for the structuring of distributed system specifications.

3.5.3 Research into LOTOS applications

In section 3.3 we cited some of the applications of LOTOS to date. In this thesis we add to the accumulated LOTOS application knowledge, by reporting on our application of (Extended) LOTOS to the CIM-OSA IIS (the case-study in chapter 5). This case study is interesting for three reasons:

- In contrast to other applications of LOTOS, the CIM-OSA IIS is not a symmetric, layered communications architecture like OSI, and the CIM-OSA IIS is more applied and specialized than the very general ODP architecture.
- The case study provides an insight into the advantages of specifying a distributed system using a pre-defined framework of formalised architectural concepts (developed in chapter 4).
- The case study provides an opportunity to test the (TLOTOS, PbLOTOS and PrLOTOS) extensions to LOTOS.

3.6 Summary

This chapter introduced formal specification languages, specifically LOTOS, as context information for the work in this thesis.

We began by listing the general attributes required of a language for the specification of distributed systems. Then we classified and overviewed specification languages in general, before focussing on the FDT LOTOS. We pointed out the inadequacies of LOTOS for the specification of quantitative timing, probabilistic and priority concerns, and resolved to remedy these inadequacies in chapters 6, 7 and 8. Also, we took a brief look at system development using LOTOS and pointed out the importance of development methods which, without relinquishing formality, allow developers to proceed in a natural manner. Finally we stated that regarding LOTOS, this thesis contributes to the areas of LOTOS language enhancement, architectural concepts, and LOTOS application.

Chapter 4

Formalizing architectural elements of distributed systems

In this chapter we build an infra-structure of architectural elements for distributed systems specification. The conviction of this chapter is that specification and design ought to be architecture-driven, rather than description language driven. The infra structure supports the architecture driven specification of distributed systems.

On the premiss that systems architecture is a special type of description, we begin from a general perspective, establishing what constitutes "description". We then become more focused to look at systems description, and form a hierarchy of the architectural elements which we consider important for general distributed system design. We base our hierarchy of architectural elements on *principles for description*. Ascending through the hierarchy we find *fundamental description ingredients*, *basic architectural ingredients*, *architectural tools and structuring concepts*, *common architectural components* and, at the top, *specific architectural components*.

We suggest XL representations and graphical denotations for elements within the infra-structure, and relate the infra-structure of architecture to a selection of existing architectures (ODP, ANSA and OSI).

In this chapter, the definitions concerned with performance use the XL features developed in chapters 6, 7 and 8. The work in this chapter provides the basis for our case-study, the formalization of the CIM OSA IIS reference architecture (chapter 5).

4.1 Introduction

We regard an architecture as a set of structuring concepts particularly suitable for the description of a specific class of problem or solution. Architectural concepts and components are elements of an architecture. Architectures are deliberately restricted in the way in which they can describe things in order to guide their users along a particular design trajectory, towards a well engineered final product. This definition of architecture is necessarily vague, since "architecture" may have subtly different meanings to different people.

Architectures exist for dealing with a wide range of systems. Some may be aimed at social or economic systems, others at technological systems such as molecular engineering, car construction, or computing systems.

An architecture is often the result of organizing practical experience into a sound engineering recipe or discipline. A good architecture embodies previous knowledge about constructing things (in the class of particular interest), implicitly imparting this knowledge to guide its users.

For computing systems, architecture is frequently cast as a set of description ingredients and tools. The system builder can use such description ingredients to build a well formed model of the problem, and apply the description tools to direct transformation of the problem-oriented description into a target solution.

4.2 The nature of description

In this section we attempt to establish the nature of description — given we are interested in architecture which is really just a specialized form of description. We take a brief look at description in general before focusing on (information) systems description. We end by isolating what we believe to be the fundamental concepts for systems description, and examine their roles.

4.2.1 What is description?

What does *to describe* mean? What is *a description*? Usually we understand the verb to mean *to communicate*, *to represent* or *to portray in an understandable way or language*, and the noun to mean *a representation*, *a specification* or *an expose*. Examining such everyday definitions we can deduce two important notions about description:

- Description is inherently indirect: a description is not actually the thing itself but merely a model or representation of it[1].

- For a description to be worthwhile it must be communicable and analysable. It might be said that these two properties are implicit in description given its definition, but we explicitly emphasis, for clarity, the requirement that descriptions are communicable between individuals and comprehensible to these individuals.

[1]Although it can be argued that such representation is really all we can know of reality anyhow.

Our ability to represent (describe), analyse, manipulate and communicate information and ideas is fundamental to the advancement of our knowledge. Many different languages, in a number of forms (natural, synthetic, textual, graphical, audible, etc.), have evolved to this end. Languages differ in their ability to describe types of information. For example, language useful for the description of geometry or shape is unsuitable for the description of temperature, odour or colour. In this chapter we are concerned with the identification and investigation of concepts useful for describing the functional, logical, performance oriented, non-physical characteristics of systems.

This discussion of fundamentals may appear obvious, and to some extent specious, but the importance of first establishing a philosophical foundation should not be disparaged. It provides the widest possible perspective in which to reason about the issue in question, promoting clarity and rationality.

4.2.2 Principles for description

We feel that this is an appropriate point to introduce *principles* or *criteria* for quality description and design. Of course quality is such a general, subjective, aesthetic, ethereal notion that it impossible to define in any sort of concrete terms. Nevertheless, many authors have proposed (sometimes contradictory, but often useful) rules of thumb or heuristics for quality in description.[3] These principles are applied to the architectural ingredients discused later in this chapter, rather than being actual ingredients themselves.

The following list of principles is by no means exhaustive, nor is it ordered in any significant way. No attempt is made to define the relationships between these principles — they may (partially) contradict or agree, or be completely orthogonal to one another, dependent upon what they are applied to and "their orientation".

Before listing some *metrics for quality*, we acknowledge [Pir91] for collecting several of these principles.

4.2.2.1 Def.P1. Parsimony

We should not introduce anything into a description which is unnecessary for our purpose — be economical with concepts. This principle is often discussed under the guise of *Occam's Razor*: "No more things should be presumed to exist than are absolutely necessary".

4.2.2.2 Def.P2. Generality

This principle demands no unnecessary restrictions. In practice it often means that we identify aspects common to different part of a system, describe these aspects, then treat their descriptions as generic characteristics of more specific things in the system. The more specific things can then be described as suitable conjunctions of generic aspects and specific characteristics. This principle leads to *reuse* instead of duplication.

[3]Some people will regard such rules of thumb for quality as (implicit) *common-sense*.

4.2.2.3 Def.P3. Orthogonality

We should not link what is independent. In practice this principle leads to a "separation of concerns".

4.2.2.4 Def.P4. Open-endedness

This rule implies that a description should be easy to maintain (extend or modify in the future). If a system is faithful to this principle, then we could extend its functionality or repair its faults at a fraction of the price of constructing a revised system from scratch.

4.2.2.5 Def.P5. Precision

This principle says that our description should not be wrong or contain unnecessary detail, so that we may correctly reason about it for a particular purpose.

4.2.2.6 Def.P6. Completeness

This principle requires that all aspects which are relevant for a particular purpose be included in the description.

4.2.2.7 Def.P7. Consistency

Do not be unnecessarily irregular or non-uniform. Some rule or convention should be adopted for creating and transforming descriptions. A heuristic for faithfulness of a description to this rule is that with a partial knowledge of the description it should be possible to hypothesize the missing parts.

4.2.3 System description

Having looked at some fairly general ideas about description we now orient our thoughts to a more specific area: the description of logical structure and function. We establish what it means to "model" or describe real world systems. Then we list and characterize the fundamental ingredients for systems description.

4.2.3.1 Modelling the real-world: phenomena to events

Model construction and analysis is synonymous with our ability to conceive and reason about the real world. First our senses perceive real world phenomena; then we sub-consciously abstract and structure this data into consciously recognizable notions and concepts about which we can reason. It is this sub-conscious process of abstraction and structuring perceived phenomena, an implicit capability of our own brains, upon which we found our ideas for explicit, conscious model construction (description).

Our ability to perceive real world 'happenings' underlies our ability to consciously conceive and model. Perceived happenings are termed phenomena — real world occurrences measured and related to our consciousness by our senses.

Often we talk about the functional aspects of a system in terms of its observable behaviour or characterizing phenomena. For example, we might describe a toaster as something which participates in the events *untoasted bread in*, *toasting bread*, and *toasted bread out*. Notice the use of the term *event* — we use the term event to mean the *most primitive* phenomenon of which we are aware at a given level of consciousness or abstraction. What we know as an event at one level of consciousness will be an abstraction for a multitude of phenomena known at another level of consciousness.

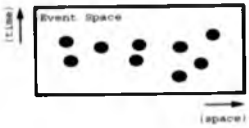


Figure 4.1: The real world as a set of potential events

Thus we conceive an actual real world system in operation as a set of events, and can model the behaviour of a real world systems as a set of possible events. Figure 4.1 depicts the real world as a set of (potential) events distributed in time and space. Events are denoted by the dark circles within the 'event space' which represents the real world.

In the following subsection we introduce the fundamental ingredients of description which allow us to communicate and analyse real world models as sets of events (or 'things', in the more general terminology of the next subsection).

4.2.4 Fundamental description ingredients

Before proceeding to investigate more specific description concepts — specification architectures — we concur with the view of ANSA [Toc90] in summarizing what we believe to be the set of fundamental description ingredients for systems description. These are: *naming*, *things*, *(de)composition* and *abstraction*.

4.2.4.1 Def. F1 Naming

A name is a symbol (normally system descriptions are in readable forms) which is used to identify something (see the named things in figure 4.2). We can describe and communicate only things which can be identified. Identifiers in computing usually take the form of meaningful strings of characters (e.g. "loop-counter", "response") or mathematical letters (e.g. "x", "∞", "Ω").

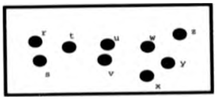


Figure 4.2: A world of named things

The ranges or classes of things to which a description language is applicable is dependent on its ability to name those things. A description language should be able to name (in a direct way) those things of concern to the user.

4.2.4.2 Def. F2 Things

ANSA [Toc90] provides the following definition of things: "Any *thing* which can be named is a *thing*; every *thing* can be named." The mapping between names and things need not be one-to-one — a name may identify a collection of things, or a thing may have several aliases.

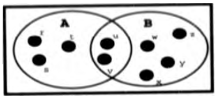


Figure 4.3: Partitioning of the world into things *A* and *B*

The division of a system into things is a subjective matter — dependent on the objectives of the process (e.g. figure 4.3). Any system may be partitioned into things in a number of distinct ways; each distinct partitioning may be useful for a particular purpose. Members of the set of partitionings may be regarded as complementary views of the same system.

A description language should facilitate complementary, distinct partitionings of a system into things. Implicit in this requirement is the fact that the description language user is able to make distinctions between things.

4.2.4.3 Def. F3 (De)composition

A single thing may be partitioned (decomposed) into a collection of 'smaller' things (e.g. figure 4.4). A single thing may also form one member of a collection (composition) of things which constitute a 'larger' thing. A single thing may be decomposed into a number of distinct decompositions; similarly a collection of things may be composed into a number of distinct compositions.

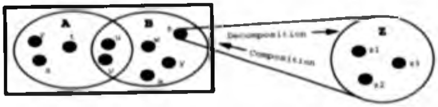


Figure 4.4: (De)composition of thing Z

The description language should provide operators for (de)composing things in ways which are of interest and convenient to the user.

4.2.4.4 Def. F4 Abstraction

Abstraction is the process of disregarding details of a description which are unimportant for some purpose. An abstraction is the result of the process of abstraction. Abstraction is an essential concept because it is impossible to describe any real-world system (thing) in perfect detail. Many different abstractions of a particular thing may exist (e.g. figure 4.5); any one of these abstractions may be more useful for a particular purpose in comparison to the other abstractions.

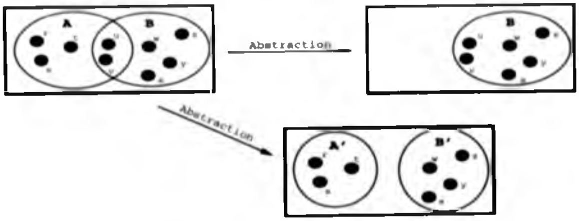


Figure 4.5: Two possible abstractions of the world

The differences between abstraction and decomposition are of a similar nature to the differences between filtering and magnification, respectively.

The concept of a ***sufficiently precise definition*** for a given purpose, should be applied as a guage when performing abstraction. Reasoning about a system, based on an abstraction of the system, has the potential to be correct if the abstraction is not ***precisely*** correct, but ***sufficiently*** correct for the intended purpose.[3]

[3] In chaos theory, no abstraction is considered sufficiently precise to correctly model the system. Chaotic systems are characterised by infinite sensitivity to initial conditions. Therefore, even if the laws governing a chaotic system can be precisely represented, such a system cannot be modelled without defining its initial state to an infinite precision, which is impossible.

A description language should be able to express abstractions of things.

4.3 Overview of existing architectures

A number of architectures for distributed systems already exist (see section 2.3). In this section we overview a selection of existing architectures, focusing on some of the more important concepts. In later sections we modify and integrate many of the existing concepts into our own hierarchy of architectural elements, and investigate formalizing this resultant architecture.

We choose three widely known architectures for distributed systems: ODP (and DAF), ANSA/ISA and OSI. ODP is the most generic of these architectures; ANSA is more applied than ODP; and OSI deals specifically with communications standards (see chapter 2 for introductions to these projects).

To avoid the unnecessary repetition of similar definitions of architectural components, drawn from ODP, ANSA and OSI, we structure the subsequent sections as follows. We give a brief introduction to the gross classifications used in each of the architectures under review. We then provide our own classification structure within which we detail the architectural concepts from the three architectures under review. For each architectural concept discussed, if any of the three existing architectures differ significantly or have interesting additional definitions we mention these.

Note: In the following discussion the meanings given to terminology often severely overlap — this is mostly unavoidable given the interdependency between these ideas and the difficulty of succinctly expressing these concepts in natural language (and, to some extent, the many different groups and cultures responsible for originating the work).

In their documentation, ODP, ANSA and OSI divide their architectures into gross classifications, for example: "modelling concepts", "architectural concepts" and "specification concepts". We endeavour to be faithful in preserving these classifications when surveying any one project, but warn the reader that these distinctions are blurred across project boundaries and can therefore be confusing.

4.3.1 The ODP architecture

For our concerns, the ODP world divides into three main categories: *basic modelling concepts*, *architectural concepts* and *specification concepts*. ODP express these concepts in carefully phrased natural language, and mathematic notation [ISO92b, ISO93a, ISO93b].

- Basic modelling concepts are those necessary to describe ODP systems and to discuss distribution. These identify essential elements of the parts of the real-world which are of interest. ODP establishes suitable abstract representations for these elements which allow us to express relations between the elements and reason about them. Basic modelling concepts include: *action*, *object*, *behaviour*, *interaction*.

- Architectural concepts are structuring concepts which are used when constructing a model of an ODP system from the basic modelling concepts with the aid of the specification concepts. Architectural concepts build upon the more fundamental basic modelling concepts. These concepts emerge when considering the issues of the problem area. Architectural concepts include: *object groups*, *domains* and *configurations*, *transparency properties* (e.g. *location*, *replication*, *fault*, *address* etc.), *causality relations* (e.g. *client* and *server objects*), *existence* (e.g. *encapsulation*, *creation and deletion* of *objects* and *classes*), *establishment* (e.g. *binding*, *trading*), *security policies*, *management architecture*.

- Specification concepts are related to the features required from a supporting specification language. ODP defines an adequate supporting specification language as one which can directly support representation of the basic modelling concepts and, at least indirectly, support the architectural concepts. Specification concepts include: *composition*, *refinement*, *trace*, *template*, *type*, *class*, *inheritance*, *polymorphism*.

4.3.2 The ANSA architecture

ANSA build their architecture on their *basic concepts* of *alphabet*, *object*, and *interface*. These are defined by natural language explanations, sometimes accompanied by set theory and a graphical syntax.

ANSA use their basic concepts to define the concepts within the other categories of their architecture. Below, we list some of these other categories, without further explanation, to give a flavour of the ANSA architecture.

- *Interactions* which include: *(in)determined interaction*, *conflict*, *composite interaction*.
- *Objects* which include: *complex object*, *undetermined object*, *role*, *agent*.
- *Specialized interactions* which include: *announcement*, *call and reply*, *non-atomic action*.
- *Structure* which includes: *multi-way join*, *directed join*, *configuration*, *multiplexer*.
- *Arrangements of objects* which include: *federation*, *hierarchy*, *client-server pair*, *peer-to-peer*.
- *Models* which include: *interpreter*, *transformer*, *policy*, *class description*.
- *Binding* which includes: *representation of structure*, *dynamic binder*.
- *Resource management* which includes: *resource manager*, *factor*.
- *Trading* which includes: *interface description*, *negotiation*, *trader*.
- *Transparencies* which include: *location*, *replication*, *migration*, *fault*.
- *Technology* which include: *infrastructure*, *information processor*.

- *Transformation of objects* which includes: *decomposition, refinement, abstraction.*
- *Miscellaneous concepts* which include: *library, directory, epoch, system.*

4.3.3 The OSI architecture

OSI architectural concepts are less generic than those of either ODP or ANSA. This is because they are specifically aimed at describing layered communication systems. The following classification of OSI architectural concepts is defined in [Tur87].

Meta-level concepts: These are used when describing OSI architectural concepts. Meta-level concepts include: *abstraction, composition, information, action, activity, interaction*, and *interaction point.*

Static concepts: These OSI concepts are concerned with static, structural or data-oriented aspects. Static concepts include: *service access point, endpoint, service primitive, protocol data unit*, and *service data unit.*

Dynamic concepts: These OSI concepts are concerned with dynamic, temporal or behaviour-oriented aspects. Dynamic concepts include: *protocol entity, protocol, service, service user, association, multiplexing*, and *splitting.*

4.4 An infra-structure of architectural elements

An infra-structure of architectural elements In the previous section we briefly surveyed three existing architectures. We now plagiarize many of the elements from these architectures and place them within our own architecture framework for distributed systems.

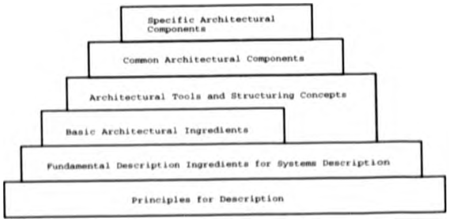

Specific Architectural Components
Common Architectural Components
Architectural Tools and Structuring Concepts
Basic Architectural Ingredients
Fundamental Description Ingredients for Systems Description
Principles for Description

Figure 4.6: Our pyramid of architecture

Figure 4.6 gives a gross perspective on the architecture framework which we now define. Our whole architecture is based upon the four *fundamental description ingredients* (section 4.2.4) immersed in the *principles for description* (section 4.2.2). Upon this foundation we construct the *basic architectural ingredients* which define the concepts which we need if we are to talk about distributed systems in general. The *architectural tools and structuring concepts* give us means to manipulate and reason about systems using the basic architectural ingredients. Using these tools and structuring concepts we can build the more complex *common architectural components* which include concepts found in most distributed systems. The apex of our pyramid, the *specific architectural components*, contains components built for specific problem domains.

We give below a list, by no means exhaustive, of architectural ingredients and concepts. Again, due to the interdependency between the following elements, we apologise for referencing elements before actually defining them.

Formalising architectural elements 'Formalizing architecture' involves giving architectural elements a formal semantics. To do this we might define a mapping between the architectural concepts to (Extended) LOTOS[4] concepts. Although such a formal mapping may be possible to define for some of the simpler architectural ingredients, this feat seems impossible for many of the higher order architectural elements. These higher order elements are of much too general a nature to tie down to (restricting) formal models. Instead we suggest *possible* mappings of some of the architectural elements to XL elements. These mappings should be treated as tentative guidelines or examples. Much more practical experience of the application of XL to formalizing architectures is required. The case study in chapter 5 takes us a step in this direction.

Figure 4.7 shows that a set of possible mapping relations that exists between architectural concepts and formal language concepts. Some of the architectural elements (especially the basic architectural ingredients) we map to XL syntactic structures, and other elements (especially the architectural tools) we map to theories concerning XL (e.g. equivalences, congruences, etc.).

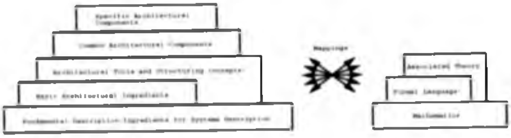

Specific Architectural Components
Common Architectural Components
Architectural Tools and Structuring Concepts
Basic Architectural Ingredients
Fundamental Description Ingredients for Systems Description
Mappings
Associated Theory
Formal Language
Mathematics

Figure 4.7: Mapping between architectural concepts and formal language concepts

Simplicity and *directness of expression* are two criteria which we use as guides when mapping architecture elements to XL. Our mapping suggestions follow.

[4]In the remainder of this chapter we will talk about *mapping to LOTOS* when we are discussing architectural elements which can be adequately represented in Standard LOTOS (and, by inclusion, also XL); and talk about *mapping to XL* when discussing architectural elements which require the enhanced expressiveness of XL.

4.4.1 Basic architectural ingredients

These are the elements which form the basic building blocks of an architecture for the description of distributed systems.

4.4.1.1 Def. B1 Event

Informal description Events are indicators of the occurrence of real-world phenomena or actions (see section 4.2.3.1). An event is atomic and instantaneous. An event occurs at a location in space and time.

Symbol, in ANSA *basic concepts*, is a similar concept but ANSA *symbols* may be decomposed. We do not consider an event as decomposable, but the real-world action which it represents may alternatively be represented as a set of other events. The reason for this seemingly unnecessary distinction is that we consider events to occur at points in time and space. It seems counter-intuitive to decompose an event occurring at a specific point in time and space into a set of events which occur at a number of distinct points in time and space. This gives us another insight into the difference between the natures of abstraction and (de)composition. We cannot compose a set of events (distributed in time and space) into a single event, but we can abstract such a set of events to a single event which characterizes the original set in some way. The characterization we choose might lead us to pick an event, from the original set, which occurs at a location in space which can be considered to be the 'typical' location for all the events in the original set. More often though, the original set of events will represent an action, and we will abstract by picking the event whose location in time represents the end of the action.

Formal representation An event is an elementary concept in LOTOS. Events are associated with transitions between states. A set of LOTOS events can be regarded as representative of some action.

4.4.1.2 Def. B2 Location

Informal description A point in time or space at which an event may occur. The location of an event (in time or space) can be described ("ordered" using some relation) only relative to other event locations.

Often though, we choose reference locations and establish event locations relative to these. Then for convenience, we assume these reference locations to be in some sense implicit, and establish the "absolute" locations of events (i.e. defined relative to the implicit reference locations).

Formal representation In LOTOS, events (except i events) are labelled with LOTOS *gate* identifiers. Optionally, events may be further annotated with event parameters.

We may use combinations of LOTOS gate identifiers and event parameters to represent location information, e.g.:

Using gate identifiers to represent location

*l*1	an event at spatial location *l*1
*t*1	an event at time location *t*1
*l*1*t*1	an event at spatial location *l*1 and time location *t*1

Using event parameters to represent location

g!*l*1	an event at spatial location *l*1
g!*t*1	an event at time location *t*1
g!*l*1!*t*1	an event at spatial location *l*1 and time location *t*1

The advantage of using event parameters (rather than gate identifiers) to represent location information is that we can treat this information as a *first class citizen*. We can pass this information around, manipulate it, create and destroy it. We can also define ordering relations over location information and hence define measures[5] for time (location in time) and, less useful in information systems modelling, measures for location in space.

Standard LOTOS has the built-in ability (through, for example, its sequential composition operator) to relatively order events in time. However it lacks any built-in mechanisms for representing and manipulating quantitative time. XL extends LOTOS with the necessary mechanisms for handling quantitative time — this topic is explored in depth in chapter 6.

XL has in built relations and operations for reasoning about and manipulating quantitative time information. We can use XL's built in quantitative time features to represent location in time, e.g.:

Using quantitative time parameters to represent location in time

g{*setLE*(3)}	an event at a time location less than or equal to 3
g?*tz* : *TimeSort*[*tz gt* 5] {*setEQ*(2) *Union setEQ*(5) *Union setGT*(*tz*)}	an event at a time location 2, 5 or greater than *tz* (de-referenced). (The event *g* is offered only when its *selection-predicate* is satisfied and when its *time-offer* is satisfied. The *time-offer* of event *g* is satisfied at times 2, 5, and greater than *tz*, de-referenced. Explained in section 6.5.1.6)
g{*setInterval*(4,9)} **ALAP**	an event at a time location *as late as possible* within the time range 4 to 9 inclusive
g **ASAP**	an event at a time location *as soon as possible*

4.4.1.3 Def. B3 Potential (Probability)

Informal description A potential event is an event with a non-zero probability of occurrence. (In figure 4.1 we represented the real world as a set of *potential* events.)

[5]Reference points, and functions (such as ordering functions) over these.

When we talk about the *potential of an event* we mean the probability of the event occurring.

Formal representation XL has built-in features for describing the probabilities of event occurrences (event potentials). These probability features are explored in depth in chapter 7. XL uses the right-associative, binary *p-choice* operator $[= \mu]$ (where μ is a probability value) for specifying the probability ratio between two mutually exclusive event sequences (traces). Some simple examples are:

$(a;\ldots)[= 0.5]$(**stop**)	event a will occur with a probability of 0.5
$(a;\ldots)[= 0.1](b;\ldots)$	events a and b will occur with probabilities of 0.1 and 0.9, respectively
$(a;\ldots)[= 0.2\,;]((b;\ldots)[= 0.125\,;](c;\ldots))$	events a, b and c will occur with probabilities of 0.2, 0.1 ($= 0.8 \times 0.125$) and 0.7 ($= 0.8 \times 0.875$), respectively

4.4.1.4 Def. B4 Object

Informal description An object is a unit of structure; it is a "seat of activity" and constrains the occurrence of events. An object's location in space is the union of locations of all events which it has the potential to constrain.

An object may be considered in a more 'physical' way (as compared to the rather meta-physical 'set of constraints' explanation above) as something which encapsulates a state and behaviour — an autonomous subsystem which interacts with its environment via well defined interfaces.

The above definition is both sufficiently descriptive and sufficiently loose for us to recognise that *our* objects may be realized as object-oriented programming languages objects [Mey88b], specification languages objects [vH89] and distributed operating system objects [ROS89b].

ODP *basic modelling concepts* similarly define an object as a "self-contained part of a system". ANSA *basic concepts* similarly defines an object as a focus of or seat for activity in a system.

Formal representation In LOTOS we can use behaviour expressions as units of structure which embody event constraints. If an object displays (parameterized) recursive behaviour we accommodate this by wrapping behaviour expressions in (parameterized) recursive process definitions[6], e.g.:

```
process Object_X[x1,x2](history:HistorySort): noexit :=
        x1 ? request:ClientRequestSort; (* a request *)
        x1 ! aFunction(request,history); (* the response *)
        Object_X[x1,x2](Update(request,history)) (* recurse *)
    []
```

[6]In the remainder of this section we use the term *process* to mean a LOTOS behaviour expression which may, or may not, include the syntactic wrapping of a LOTOS process.

When we talk about the *potential of an event* we mean the probability of the event occurring.

Formal representation XL has built-in features for describing the probabilities of event occurrences (event potentials). These probability features are explored in depth in chapter 7. XL uses the right-associative, binary *p-choice* operator $[=\mu]$ (where μ is a probability value) for specifying the probability ratio between two mutually exclusive event sequences (traces). Some simple examples are:

$(a;\ldots)[=0.5](\mathbf{stop})$	event a will occur with a probability of 0.5
$(a;\ldots)[=0.1](b;\ldots)$	events a and b will occur with probabilities of 0.1 and 0.9, respectively
$(a;\ldots)[=0.2\ ]((b;\ldots)[=0.125\ ](c\ \ldots))$	events a, b and c will occur with probabilities of 0.2, 0.1 ($= 0.8 \times 0.125$) and 0.7 ($= 0.8 \times 0.875$), respectively

4.4.1.4 Def. B4 Object

Informal description An object is a unit of structure; it is a "seat of activity" and constrains the occurrence of events. An object's location in space is the union of locations of all events which it has the potential to constrain.

An object may be considered in a more 'physical' way (as compared to the rather metaphysical 'set of constraints' explanation above) as something which encapsulates a state and behaviour — an autonomous subsystem which interacts with its environment via well defined interfaces.

The above definition is both sufficiently descriptive and sufficiently loose for us to recognise that *our* objects may be realized as object oriented programming languages objects [Mey88b], specification languages objects [vH89] and distributed operating system objects [ROS89b].

ODP *basic modelling concepts* similarly define an object as a "self-contained part of a system". ANSA *basic concepts* similarly defines an object as a focus of or seat for activity in a system.

Formal representation In LOTOS we can use behaviour expressions as units of structure which embody event constraints. If an object displays (parameterized) recursive behaviour we accommodate this by wrapping behaviour expressions in (parameterized) recursive process definitions[6], e.g.:

```
process Object_X[x1,x2](history:HistorySort) : noexit :=
        x1 ? request:ClientRequestSort; (* a request *)
        x1 ! aFunction(request,history); (* the response *)
        Object_X[x1,x2](Update(request,history)) (* recurse *)
    []
```

[6]In the remainder of this section we use the term *process* to mean a LOTOS behaviour expression which may, or may not, include the syntactic wrapping of a LOTOS process.

```
        x2 ! Terminate:ManagementCommandSort; (* terminate command *)
        stop (* terminate *)
endproc (* Object_X *)
```

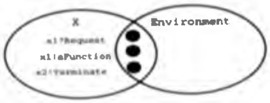


Figure 4.8: Object_X

This example describes an object which is willing to either perform a request, aFunction and then recurse, or accept a command to terminate. Figure 4.8 shows object X constraining the three events.

4.4.1.5 Def. B5 Environment

Informal description An object's environment is all parts of the universe of discourse which are not part of the object.

Formal representation The concept of environment is implicitly supported by LOTOS — the environment of a process (object) is everything which is not a part of that process (object).

4.4.1.6 Def. B6 Interaction

Informal description This term is used to denote an event (or set of events when an interaction is defined to span more than a single event) in which two or more objects participate (alternatively, two or more objects can be said to be 'constraining' the event). A single event interaction represents a synchronization between its participating objects.

ODP *basic modelling concepts* and ANSA *interactions* similarly define interaction. However, in ANSA *interconnection points* (or *connections* as termed by ANSA) can exist only between pairs of objects, so that in ANSA only two objects may synchronize on any single event.

Formal representation In LOTOS, an interaction is defined as the occurrence of an event which is constrained by two or more processes. This nicely fits with our architectural concept of interaction, except that for architectural purposes we allow an interaction to be a set of such events. The following example illustrates an interaction between two (unnamed) objects.

```
           (i;interact1;interact2;exit)
|[interact1,interact2]|
           (interact1;i;interact2;exit)
```

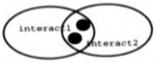


Figure 4.9: An interaction between two unnamed objects

The interaction(s) between the two unnamed objects (processes) consists of the events *interact1* and *interact2*.

4.4.1.7 Def. B7 Interaction Point

Informal description The location in space at which an interaction may occur.

ODP *basic modelling concepts* give the same definition. ANSA allow their *connections* to be named. OSI calls this concept a *service access point* (although this really represents a more specific concept).

Formal representation The interaction points of a process are identified to be the locations of the events which occur at observable gates of the process. For example, the interaction points of the process X in figure 4.8 are the locations of all events which occur at the gates $x1$ and $x2$.

Often, we will require to make a distinction between an interaction point (i.e. location in space) and the actual events which occur at the interaction point. For this we must attribute event denotations with two distinct labels, e.g.:

```
    interaction_point_id ! event_id
or
    event_id ! interaction_point_id
```

4.4.1.8 Def. B8 Data

Informal description Symbols, or a set of event occurrences from which information may be derived. This means that we can consider an event occurrence to be associated with a set of symbols, or a particular configuration of event occurrences in time and space as conveying some information.

Formal representation Any identifier in a LOTOS description may be used to represent data.

4.4.1.9 Def. B9 Communication

Informal description This is the passing of data via an interaction.

ODP *basic modelling concepts* define communication to be a "sequence of causally ordered interactions between two or more objects, which results in conveyance of information between them". ODP and ANSA suggest a number of events to be communication. ANSA orders symbol sequences occurring over *connections* to represent communication.

Formal representation This is the 'passing' of data via an interaction. The data may be encoded in a LOTOS gate name, e.g. *request*, *do_it*, etc. Then, to communicate such data, the objects participating in the communication must synchronize on these events, e.g.:

```
(request; ... ) |[request]| (i; request; ... )
```

More often though, we use LOTOS event parameters to encode data, e.g. $g!data$, $g!request$, $g?num : Nat$, etc. This method has an advantage that it is possible to negotiate values between interacting (communicating) objects. In the following example:

```
(g?x:Nat[x gt 4]; ...) |[g]| (i; g?x:Nat[x lt 6]; ...)
```

the two unnamed objects interact (synchronize) to negotiate $x = 5$. Unless we label interactions in terms of direction, we can only say that the data value 5 is 'communicated' to both objects; we cannot attribute a direction to this communication.

4.4.1.10 Def. B10 Behaviour

Informal description The behaviour of an object is the set of all sequences of events in which the object may participate. Such sequences are subject to the constraints which the object itself imposes on event sequences, and subject to any constraints which the environment imposes on the events in which the object participates.

Formal representation If we use a LOTOS 'process' to represent an object, then the behaviour of an object will be the set of all possible event sequences (traces) in which that object may participate. For example, the behaviour of the object:

```
x; (y; exit [] z; exit)
```

is the set:
$\{x \rightarrow y \rightarrow exit, x \rightarrow z \rightarrow exit\}$

4.4.1.11 Def. B11 State

Informal description The state of an object is defined to be the data embodied by it. This (together with the state of the object's environment) will determine the future behaviour of the object. The state of an object can change only as a result of actions internal to the object, or interactions between the object and its environment.

Formal representation The state of an object (at a point in time) is anything embodied by it (at that point in time) which may affect the future behaviour of the object. For a LOTOS process, this identifies something like a "state vector" which includes the static structure of the behaviour expression, the values of all incorporated dynamic data, and an indication of the current state(s) reached in the execution of the behaviour expression (at the particular point in time).

4.4.1.12 Def. B12 Internal Event

Informal description These are events which are only (directly) constrained by a single object. (However, sub-objects of an object X may constrain an event which is considered internal with respect to the object X, but observable with respect to the sub-objects). Claiming an event is internal to a particular object gives that object direct control over the occurrence of that event. This is useful in a description if we want to show that an object has sole responsibility for some action.

Formal representation In LOTOS, gates at which internal events may occur are identified by the **hide** operator, or by the reserved event name **i**. Thus, in the following process definition, events at gates a, x and the **i** events are internal events with respect to process P.

```
process P[b,y] : exit :=
   hide a, x in ...; i;...; a;... ; i; ... ; y; ...
endproc (* P *)
```

4.4.1.13 Def. B13 Actions

Informal description An action is a sequence of events, or just a single event.

Formal representation Actions are expressed as sequences of LOTOS events, or as single LOTOS events.

4.4.1.14 Def. B14 Parallelism for Actions

Informal description If we consider an action as a sequence of events, then two or more actions are said to occur in parallel (i.e. their durations of occurrence overlap) if their event sequences are interleaved.

Formal representation Parallel actions are concurrent sequences of events. Note that in our architecture we talk about the *concurrency of events* and *parallelism of actions*.

4.4.1.15 Def. B15 Interface

Informal description An interface is a set of interaction points. Normally we associate an interface with an object — "the interface between an object and its environment". In such a case, both the object and the environment are responsible for 'shaping' the interface, i.e. the conjunction of the constraints imposed by the object and by the environment are, alone, responsible for creating and characterizing that interface.

Formal representation An interface of an object defines how it constrains a set of interaction points.

Normally an interface will be defined by:

- its location — the locations of the events occurring at the interaction points which constitute it — see Def. B2
- the format of the data communicated at the interface — see Def. B8
- some behavioural properties — see Def. B10.

4.4.2 Architectural tools and structuring concepts

Architectural elements under this heading can be applied to the basic architectural ingredients in order to build higher order architectural elements.

4.4.2.1 Relations between descriptions

Informal description Developing a system usually entails moving from one description D_m of the system to another D_n which is further along the design trajectory. The developer will compare two such descriptions of a system using a number of given relations which are useful in guiding or assessing the development process.

Examples of relations include:

D_m **absts** D_n	D_m is an abstraction of D_n
D_n **decps** D_m	D_n is a decomposition of D_m
D_n **impls** D_m	D_n is a implementation of D_m

Only fuzzy distinctions exist between informally defined relations such as the ones above.[7] Often a development step will involve using a combination of relations.

The next few paragraphs overview the main categories of relations.

[7]It is not possible to formally define such general concepts but this does not preclude their usefulness

Formal representation Above we mentioned some very general relations which may exist between descriptions, e.g.:

$$D_m \ \textbf{absts}\ D_n$$
$$D_n \ \textbf{decps}\ D_m$$

In the LOTOS world, there is a growing body of work concerned with establishing useful relations, and providing methods for testing and verifying such relations.

It is not possible, or desirable, to attribute formal meanings to very general relations, such as **absts** and **decps**. However, for specific contexts, it is useful to establish a prescription of formal relations which may capture some of the properties of the informal relations. Suggestions for such prescriptions are made in the following paragraphs.

See appendix G for more details of LOTOS formal relations.

4.4.2.2 Def. TS1 (De)Composition

Informal description The activities of composition and decomposition are complementary to each other, as illustrated in figure 4.4.

The activity of composition takes a 'set of distinct things' and produces a 'single thing' (a composition). The activity of decomposition takes a 'single thing' and produces a 'set of distinct things' (a decomposition).

The 'set of distinct things' may contain things of many types, e.g. events, objects, data, constraints, composition operators, etc. Therefore, in general, the type of a resultant composition will be different from the type(s) of its constituent things.

In our architecture not everything (i.e. events — see Def. B1 events) can be decomposed. This is not aligned with ANSA's view which states that absolutely everything (in its architecture) can be decomposed — again, see Def. B1.

Formal representation LOTOS has a number of composition operators which allow us to compose a set of individual objects (processes) into a single, composite object (process). The following list contains some fairly obvious examples of composition.

```
Event_1 ; Object_1          sequential composition of event and object
Ob      _                    ject_1 >> Object_2       sequential composition of two objects
Ob      _  []       _        ject_1    Object_2       choice composition of two objects
Ob      _                _   ject_1 |[gates]| Object_2  parallel (concurrent) composition of t
                            objects which may interact via the gate set gates
Ob      _  [> Object_2      disabling composition of two objects
```

Decomposing an object involves describing that object as a set of 'smaller' objects (a decomposition). Some of the 'smaller' objects may already exist in (say) a "re-use library" others may have to be created from scratch.

(De)Composition in practice The practice of:

- forming a composition of already existing objects

- decomposing into a number of new objects

is not particularly easy. The behaviour of a composite object may not be easy to derive from the individual behaviours of its constituents. This is especially true if the objects interact with one another via 'wide' interfaces (strong coupling), or if the behaviour of a constituent object is radically affected by its new context. There exist a number of congruence relations (relations which hold true regardless of context, see appendix G) which can help when reasoning about the behaviour of a composite object.

Consider figure 4.10 which shows a very simple development of an object *obj.1*.

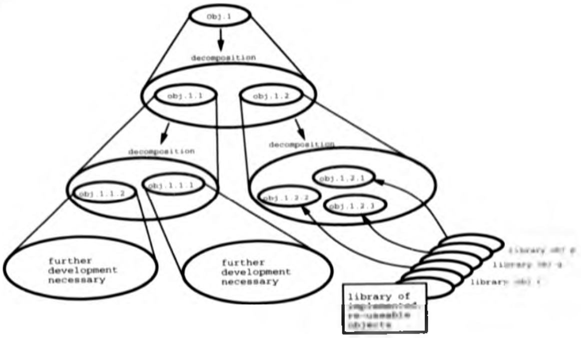


Figure 4.10: A simple development of an object

In this diagram, decomposition is used as a top-down development method, while composition is used as a bottom-up method. Object *obj*.1 is developed through decomposition into objects *obj*.1.1 and *obj*.1.2. In contrast, object *obj*.1.2 is developed through a composition of objects *obj*.1.2.1, *obj*.1.2.2 and *obj*.1.2.3 which have been discovered in the library of implemented, re-usable objects. Such a library will contain common, generic objects which can be used like pre-fabricated building blocks.

We now give a possible list of the relations which the developer might use in this development process. ('•' denotes a LOTOS composition operator.)

$$(obj.1.1 \bullet obj.1.2) \ \mathbf{decps} \ obj.1$$
$$(obj.1.1.1 \bullet obj.1.1.2) \ \mathbf{decps} \ obj.1.1$$
$$(obj.1.2.1 \bullet obj.1.2.2 \bullet obj.1.2.3) \ \mathbf{decps} \ obj.1.2$$

To add more formality and detail to this development process, we employ the help of **cext**, **cred** and $=_{tc}$, three of the formal LOTOS relations (see appendix G for further explanation).

$$(obj.1.1 * obj.1.2)\ \textbf{cext}\ obj.1'\ \textbf{cred}\ obj.1$$
$$(obj.1.1.1 * obj.1.1.2)\ \textbf{cext}\ obj.1.1'\ \textbf{cred}\ obj.1.1$$
$$(obj.1.2.1 * obj.1.2.2 * obj.1.2.3)\ \textbf{cext} obj.1.2'\ \textbf{cred}\ obj.1.2$$

The first of these equations says that object $obj.1'$ is a congruent reduction of object $obj.1$. We imagine that $obj.1$ is a specification level object which contains a certain amount of implementation freedom in terms of non-determinism. Object $obj.1'$ is a reduction of some of the non-determinism in $obj.1$; it is also a congruent reduction of *obj.1* allowing it to be placed in any context in which *obj.1* can be placed. Object $(obj.1.1 * obj.1.2)$ is a congruent extension of object $obj.1'$. We imagine that object $(obj.1.1 * obj.1.2)$ adds some additional behaviour to object $obj.1'$ — an addition which does not destroy the original behaviour specified in $obj.1'$.[8]

In order to ensure that objects $obj.1.2.1$, $obj.1.2.2$ and $obj.1.2.3$ can indeed be substituted for ready-built library objects (and hence require no further development, unlike objects $obj.1.1.1$ and $obj.1.1.2$) we must enlist the help of another formal LOTOS relation: $=_{tc}$ (testing congruence).

$obj.1.2.1 =_{tc} library.obj.p$

$obj.1.2.2 =_{tc} library.obj.q$

$obj.1.2.3 =_{tc} library.obj.r$

If $obj.1.2.1$ is testing congruent to *library.obj.p* it means that these two objects cannot be distinguished from one another by testing. Hence $obj.1.2.1$ can be substituted by the pre-defined library object *library.obj.p*.

The relationship between (de)composition and abstraction In Def. B1 we said that an event was atomic and therefore not itself decomposable. This is not really a restriction since we can further decompose the real world action which an event represents. For example, we could decompose the *toasting* action, represented by the event *toasting_bread*, into the events:

start_toasting, more_toasting, toasted

The above sequence of events represents a decomposition of the action *toasting*. Also, the original single *toasting_bread* event represents an abstraction of the above sequence of events. This leads us to ask how abstraction and (de)composition are related. We think of their relationship as follows.[9]

[8]Often, a specification will describe only *required* behaviour, but its implementation may well include *additional* behaviour for, say, handling implementation level error scenarios which are not specification level requirements.

[9]We acknowledge that the relation between (de)composition and abstraction is somewhat subjective.

We define **decps** (decomposition relation) identifications to be a subset of **absts** (abstraction relation) identifications, i.e.:

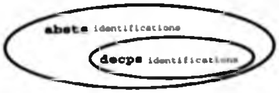


Figure 4.11: Decomposition as a subset of abstraction

We justify this by example. Consider the equations:

$$(x; y; \mathbf{stop}) \quad \mathbf{absts} \quad (x; i; y; \mathbf{stop}) \tag{4.1}$$

$$(x; i; y; \mathbf{stop}) \quad \mathbf{decps} \quad (x; y; \mathbf{stop}) \tag{4.2}$$

$$(x; y; \mathbf{stop}) \quad \mathbf{absts} \quad ((x; i; y; \mathbf{stop})|||(z; \mathbf{stop})) \tag{4.3}$$

$$((x; i; y; \mathbf{stop})|||(z; \mathbf{stop})) \quad \textbf{not decps} \quad (x; y; \mathbf{stop}) \tag{4.4}$$

Equation 4.1 tells us that one abstraction of an object is just its interface (the events x and y, in this case). Equation 4.2 complements the transformation denoted by equation 4.1, saying that an object may be decomposed to give a description of the object with some internal detail. In equations 4.1 and 4.2, **absts** and **decps** identify entities associated only with the object on which they act.

Equation 4.3 represents **absts** again acting on the object. In this case though, the **absts** transformation *generates* an entity which is rather different from the object. There is no equivalent **decps** tranformation which acts on the object, see equation 4.4.

Consider the following points which summarize our view on (de)composition versus abstraction:

- D_1 **absts** D_2 says that entities may be in D_2 which are not in D_1; or that, entities which exist in D_2 may be disregarded in D_1. And in comparison...
- D_2 **decps** D_1 says that, D_2 may only *magnify* those entities which already exist in D_1; or that, D_1 may *collapse* entities which already exist in D_2.

Since not formally founded, these definitions are still slightly imprecise, but they serve us in forming a perspective on design transformations.

4.4.2.3 Def. TS2 Abstraction

Informal description This is the suppression of irrelevant detail. An abstraction captures the essential details of more detailed description. Abstraction can be applied in the dimensions of space (e.g. describe only those things within a certain locality), time (e.g. to describe only certain epochs of a system, maybe connection-phase, data-transfer phase or installation, etc.), and functionality (e.g. to describe only things which realize particular functions).

Both ODP and ANSA define a number of 'standard' abstractions or *projections* which provide different, but complementary views on information processing systems (see section 2.3.3.2).

Formal representation Using LOTOS it is possible to describe (partial) designs at arbitrary levels and in arbitrary dimensions of abstraction. LOTOS uniformly supports abstraction in the dimensions of space, time and functionality. Consider the following description:

```
          (taskD_epoch1; taskB_epoch2; exit)
|[taskB_epoch2]|
          (taskB_epoch2; taskA_epoch3; exit)
```

An abstraction of the original description *in time*, ignoring events which occur during odd epochs, might produce:

```
          (taskB_epoch2; exit)
|[taskB_epoch2]|
          (taskB_epoch2; exit)
```

An abstraction of the original description *in time*, ignoring the relative timing between events (replacing occurrences of the sequencing operator ';' by the interleaving operator '|||')[10], might produce:

```
          (taskD; exit ||| taskB; exit)
|[taskB]|
          (taskB; exit ||| taskA; exit)
```

An abstraction of the original description *in space*, ignoring events which occur wholly within the spatial locality of the first unnamed object/behaviour expression (i.e. ignoring *taskD* because it appears in the first object only), might produce:

```
          (taskB_epoch2; exit)
|[taskB_epoch2]|
          (taskB_epoch2; taskA_epoch3; exit)
```

An abstraction of the original description *in functionality*, ignoring the functionality realized by *taskB*, might produce:

[10]Examining the resulting abstraction, a reader might ask: "The two objects still synchronise on the *taskB* event — is synchronisation not a time concern?". Our view goes like this. It is possible to just consider this a synchronisation in space. The time of synchronisation (i.e. *taskB* event occurrence) is not relevant — we can not order it relative to the other events anyway. But, if an event occurs, it must occur at some location in time, relevant or not.

```
        (taskD_epoch1; exit)
|||
        (taskA_epoch3; exit)
```

If we think of abstraction as just being applied to *tasks* in the above example, then an abstraction transformation is responsible for *generating* completely new *tasks*, in some instances, and responsible for *ignoring tasks* which originally existed, in other instances. This in in contrast to a (de)composition transformation which must work with what is already there.

4.4.2.4 Def. TS3 Transformation

Informal description In its most general sense, this concept means to take a description and modify it in some way (e.g. by adding and removing events, constraints, data, etc.) to change it into another distinct but related description.

In a more specific sense, a transformation means to take a description of a system at one level of abstraction and to alter the description (by decomposition) so that it describes the same system, at the same level of abstraction, but with magnified detail.

Formal representation We consider transformations within a design trajectory. Within this subsection we have looked at two very general transformation relations **abats** and **decps**, and a few more specific, formal, supporting relations **cred**, **cext** and $=_{R}$.

Normally we require transformations to:

- interpret 'syntactic structure' in description (e.g. |[]| — as a physical distribution operator; $P[]P$ as duplication for reliability, etc.) and develop this accordingly
- preserve the semantics of a description.

LOTOSPHERE [Pir91] captures this two fold property of transformation in what it calls a "*correctness preserving transformation relation*" (R_{CPT}). An R_{CPT} consists of two components:

- an "R_T transformation relation" (e.g. gate splitting, making parallelism explicit, making states explicit, etc. see [Pir91]) which is responsible for the interpretation and development of 'syntactic structure' in keeping with specific design goals
- an "R_{CP} correctness preserving transformation relation" (e.g. an implementation, equivalence or congruence relation — see appendix G) which is responsible for ensuring that certain semantic properties are maintained.

4.4.2.5 Def. TS4 Refinement

Informal description This is a specific type of transformation. Refinement makes a description more implementation oriented.

ANSA point out that in practice this is often achieved through two routes: the resolution of non-determinism, and the resolution of structure. A system description may be refined to another by reducing the amount of non-determinism in the description, or by increasing the amount of structural detail in the description.

Formal representation Formal LOTOS relations may be used to support the task of refinement. For example, the **red** ("reduction") relation may be used to support the resolution of non determinism, and the **ext** ("extension") relation may be used to support the resolution of structural detail (see appendix G and [Toc90]).

4.4.2.6 Def. TS5 Non-determinism

Informal description This provides the system designer with a means of expressing a set of possible descriptions.

The tool of non-determinism is often used for specifications. A *specification* is a description interpreted in a specialized way as a description of a set of *implementations*. The use of non-determinism allows specifications to compactly express, and not unnecessarily constrain, a number of possible implementations.

The ability to express non determinism is also useful in the description of a particular implementation[11]. Using non-determinism we can express partially ordered sequences of events, and hence represent concurrency (see next concept) within a system.

Formal representation LOTOS supports non determinism for use in expressing specification, concurrency or environmental influence.

Using the LOTOS *choice operator* we can express non-determinism in a specification of a system (i.e. a set of possible implementations of a system). Consider the LOTOS fragment:

i; x; **stop** [] i; y; **stop**

Interpreting this results in the behaviour trace set:

$$\{\prec x \succ, \prec y \succ\}$$

The choice between the two possible behavioural paths remains unresolved at this level of description. We could consider this description as a specification of a system, where each trace in the trace set represents a possible implementation of the system.

The value negotiation mechanism in LOTOS also provides a powerful means of expressing non determinism. Through this mechanism we can express the non deterministic choice of one value from a set of possible values. For example, in:

[11]The term implementation is used here to emphasise that we are not talking about a specification but rather the description of a single system. That is to say, here we are using non determinism as a tool within the description of a single system, whereas in the preceding paragraph, we discussed using non determinism as a tool for conveniently expressing sets of possible single systems.

```
(g ? m:Nat, ...) || (g ? n:Nat; ...)
```

the value of m or n is indeterminable (except that it is a term of sort *Nat*).

The LOTOS choice operator could be used rather indirectly for concurrency. The hallmark of concurrent activities is their independence from one another and hence the difficulty of establishing a total ordering of their events in time. Concurrent activities are often represented as non-deterministic orderings of events. The choice operator can be used to construct such non-deterministic orderings. For example, consider activities A and B to be represented by the following event sequences:

```
A := a1; a2; stop
B := b1; stop
```

If activities A and B are concurrent[12], we could express their resulting non-deterministic ordering, using the choice operator, as shown below.

```
A ||| B :=
              (a1; a2; b1; stop)
           [] (a1; b1; a2; stop)
           [] (b1; a1; a2; stop)
```

LOTOS supplies the |[]| operator to its users, saving them from having to express concurrency in an explicit manner using the choice operator.

4.4.2.7 Def. TS6 Concurrency

Informal description Events are said to be concurrent if there is no need to relate the events in time.

Formal representation In LOTOS, a set of concurrent event sequences is represented as a non-deterministic choice between all possible fair mergings/interleavings of the event sequences in the set (see section 6.3.8). Concurrency can be expressed using the |[*gates*]| operator which takes two sequences of event offers, generates all possible fair interleavings of these, and returns a non-deterministic choice between these interleavings. Pairs of matching event offers, whose gates names are in the set *gates*, must synchronise on single events.

4.4.2.8 Def. TS7 Separation of concerns

Informal description The architect can structure his design in a modular way such that all elements of any one module have a *strong cohesion* (i.e. each set of closely related aspects are gathered into a module), compared to the *weak coupling* between modules (i.e. module interfaces should be 'narrow').

[12]More technically, we talk about parallel activities and concurrent events.

Formal representation A LOTOS description can be structured as a hierarchy of processes. Each set of strongly related concerns may be assigned a process, such that only weak coupling exists between the processes, at any one level in the hierarchy.

If each process is assigned a "resource" concern, the resulting LOTOS specification style is said to be "resource-oriented" [VSvSB90] (or "object-based" [CJ92, Cla90, MC93]). If each process is assigned a "constraint" concern, the resulting LOTOS specification style is said to be "constraint-oriented" [VSvSB90].

4.4.2.9 Def. TS8 Relegation of description

Informal description This tool is known in many forms, some more or less sophisticated than others, e.g. inheritance, definition and reference, template and instantiation. Basically it allows the architecture to isolate generic aspects of the system, define these once (see next concept), reference them when required (i.e. relegate description), treating such references as the actual things themselves.

(Related to the idea of relegation of description is the idea of polymorphism/genericity. Polymorphism/genericism is a description tool which allows designers to encode algorithms such that the algorithms are applicable to any subset of data from a set[13]). Cardelli and Wegner [CW85] write with insight on this subject, categorizing various flavours of polymorphism including universal parametric, universal inclusion, *ad-hoc* overloading, *ad-hoc* coercion.)

Formal representation Relegation of description through 'reference and definition' is supported both in the ACT ONE data typing and process algebra parts of LOTOS. Relegation in ACT ONE is found under the guise of *enrichment*. ACT ONE descriptions may reference (relegate description to) ACT ONE definitions elsewhere using the **is** operator. For example, in:

```
type zType is xType, yType
—
endtype (* zType *)
```

type *zType* relegates a part of its description to the type definitions of *xType* and *yType*. In the process algebra part of LOTOS, relegation is realized through process instantiation and definition. For example, in:

```
(* Definition of process Z follows *)
process Z :=

    X (* Instantiation of X — description relegation to def. of X *)

endproc (* Z *)
```

[13]whether this set be a set of sorts or, taking the definition of polymorphism to an extreme, a set of terms. Under this extreme definition, even a + operator with domain and co-domain sorts fixed as **Natural** is considered polymorphic, since + can accept any natural number term from the (infinite) set of natural numbers.

```
(* Definition of process X follows *)
process X := ... endproc (* X *)
```

process Z defers a part of its behavioural definition to a definition (process X) elsewhere.

Relegation is not unlike concepts such as import/export lists, inheritance, instantiation and template. Several authors [May89, CRS89] have discussed how the concept of inheritance (a more specialized form of relegation) may be supported in LOTOS. They advocate either constraining the user to a particular style of LOTOS [vH89], or suggest semantic extensions [Rud92].

4.4.3 Common architectural components

These are architectural elements which are common to most distributed systems. The common architectural components exist at a higher level than the basic architectural ingredients.

To provide a handle on complexity and enhance understandability we arrange the common architectural components into a loose classification hierarchy (see figure 4.12). Also, to aid the readability of systems structured out of the architectural components we now define, we provide graphical denotations for many of these components. Our classification is not complete, nor without ambiguity, but it does provide a reasonable framework for reasoning about the structure of typical distributed systems.

The first two subclasses identified within *common architectural components* are the *synchronous combinators* and the *components*. The *synchronous combinators* are responsible for carrying synchronous communications between the *components*. *Synchronous combinators* tend not to exist as real world realizations but exist in the abstract design world as useful for combining *components*. The *components* usually exist as identifiable units of structure not only in the abstract design world but also in the real implementation world. Conceptually we think of *components* as either composed of other *components*, or (for *components* which cannot sensibly or justifiably be decomposed into other *components*) just consisting of XL text.

A criticism of our classification is that it can prove impossible to place a *component* in one particular class. Our answer to this is that *components* should be classified according to their dominant characteristic or purpose. The classification of *components* is there to provide a reasonable means of organizing a specification according to problem domain structure[14] – the classificaton will not prove perfect for each specific situation. A good heuristic for structuring a specification in terms of our *component* classes is that if a *component* has no dominant characteristic by which to classify it, then label it as an *untyped component* (Def. CC15) and then further decompose this *component* into a number of *sub-components*, each of which should reflect one of the original *component's* primary characteristics. Alternatively, if a *component* proves difficult to classify or decompose further, then just write some XL text for it.

[14]in this case, the very general problem domain of distributed computing systems

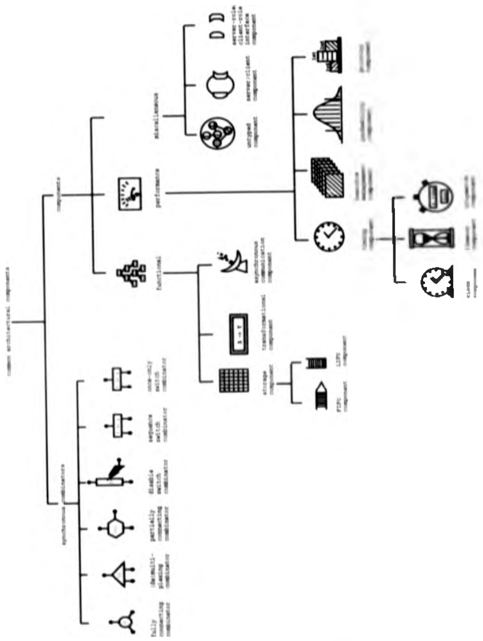

common architectural components
synchronous combinators
components
functional
performance
miscellaneous
fully connecting combinator
(de)multiplexing combinator
partially connecting combinator
disable switch combinator
sequence switch combinator
once-only switch combinator
storage component
transformational component
asynchronous communication component
unplugged component
server/client component
server-role client-role interface component
FIFO component
LIFO component

Figure 4.12: Classification of *common architectural components*

The following subsections provide details on the *synchronous combinators* and *components*, and their subclasses.

4.4.4 Synchronous combinators

A *synchronous combinator* is a means of glueing together two or more *components* such that the glued *components* synchronously communicate with one another. The *synchronous combinators* listed below are really 'packaging' for LOTOS process algebra operators. We justify this repackaging of LOTOS operators on the grounds that we want to introduce terminology and graphics suitable for discussing and depicting combinations of the *components* defined in the next subsection.

4.4.4.1 Def. CS1 Fully connecting combinator

This *combinator* connects a number of *components* so that they fully synchronise on all common (X)L gates.

The *fully connecting combinator* will often be represented in (X)L by a combination of || operators and parentheses. For example, the *components worker_A*, *worker_B* and *worker_C* in the LOTOS fragment below are fully connected. In the graphical depiction of this example (figure 4.13), the *fully connecting combinator* is depicted by the large circle. The *fully connecting combinator* is symmetrical therefore the relative positioning of the arcs joining *components* to the *fully connecting combinator* is not important.

worker_A[workplace] || (worker_B[workplace] || worker_C[workplace])

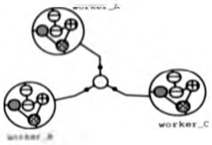


Figure 4.13: A *fully connecting combinator* joins three *components*

4.4.4.2 Def. CS2 Partially connecting combinator

This *combinator* connects a number of *components* so that they synchronise on only some of their common (X)L gates.

The *partially connecting combinator* will often be represented in (X)L by a parallel combination using the |[]| operator. For example, the *components fully_co_op_worker_A*, *partly_co_op_worker_B* and *fully_co_op_worker_C* in the LOTOS fragment below are partially connected. All three *workers* co-operate (synchronise) at *workbench1*, but only *workers A* and *C* co-operate (synchronise) at *workbench2*. In the graphical depiction of this example (figure 4.14), the *partially connecting combinator* is depicted by the hexagon. The *partially connecting combinator* is asymmetric therefore the relative positioning of the arcs connecting *components* to the *partially connecting combinator* graphic is important. Fully synchronised *components* are connected to either the top or the bottom of the hexagon. Partially synchronised *components* are connected to the side of the hexagon. Detailed gate information is suppressed in the diagrams.

(fully_co_op_worker_A[workbench1,workbench2]
|| fully_co_op_worker_C[workbench1,workbench2])
|[workbench1]| partly_co_op_worker_C[workbench1,workbench2]

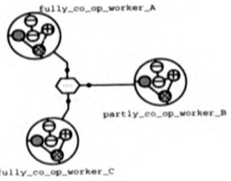


Figure 4.14: A *partially connecting combinator* joins three *components*

4.4.4.3 Def. CS3 (De)multiplexing combinator

This *combinator* connects a number of *components* such that one of the *components* (the 'primary') may synchronise with any of the other *components* (the 'secondaries'), but no synchronisation amongst the 'secondaries' is possible.

The *(de)multiplexing combinator* will often be represented in (X)L by a parallel combination using the ||| operator. For example, the *components trunk_line*, *local_line_X*, *local_line_Y* and *local_line_Z* in the LOTOS fragment below are multiplexed (with *trunk_line* as the 'primary'). In the graphical depiction of this example (figure 4.15), the *multiplexing combinator* is depicted by the triangle, with the 'primary' *component* connected by an arc to the apex of the triangle, and the 'secondaries' connected by arcs to the opposite base side of the triangle.

trunk_line[x,y,z] || (local_line_X[x] ||| local_line_Y[y] ||| local_line_Z[z])

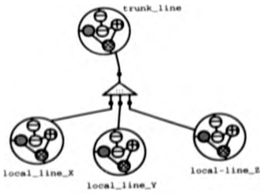


Figure 4.15: A *multiplexing combinator* joins four *components*

4.4.4.4 Def. CS4 Disable switch combinator

In its ternary form, this *combinator* connects three *components* such that the 'permanent' *component* may be disconnected from the 'disablable' *component* and permanently reconnected to the 'disabler' *component*.

The *disable switch combinator* is represented in (X)L using the ▷ operator. In the example below, *processor* is the 'permanent', *normal_code* is the 'disablable' and *exception_code* is the 'disabler'. In the graphical depiction of this example (figure 4.16), the 'permanent' is connected by an arc to the top of the rectangle, the 'disablable' is connected by an arc to the bottom of the rectangle, and the 'disabler' is connected by an arc to the zig-zagging 'lightning' graphic.

processor[memory] || (normal_code[memory] ▷ exception_code[memory])

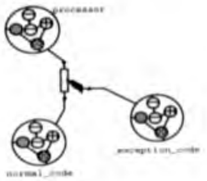


Figure 4.16: A *disable switch combinator* joins three *components*

4.4.4.5 Def. CS5 Sequence switch combinator

This *combinator* combines a number of *components* such that one of the *components* (the 'primary') can synchronise with each of the other *components* (the 'secondaries'), in a predefined sequence. Once a 'secondary' has exhausted its function, the 'primary' synchronises with the next 'secondary' in the sequence.

The *sequence switch combinator* is represented in (X)L using the ≫ operator. In the example below, *telephone* is the 'primary', while *connect*, *transmit* and *disconnect* are the 'secondaries'. In the graphical depiction of this example (figure 4.17), the 'primary' is connected by an arc to the top of the rectangle, and the 'secondaries' are connected by arcs to the bottom of the rectangle.

telephone[data] || (connect[data] ≫ transmit[data] ≫ disconnect[data])

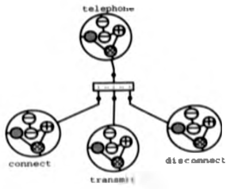


Figure 4.17: A *sequence switch combinator* joins four *components*

4.4.4.6 Def.CS6. Once-only switch combinator

This *combinator* synchronises one *component* (the 'permanent') with one of a number of 'secondary' components. Whenever the first synchronisation between the 'primary' and one of the 'secondaries' occurs, the combination of the 'primary' to this particular 'secondary' becomes a permanent arrangement and possibility of synchronisations with any of the other 'secondaries' no longer exists.

The *once-only switch combinator* is represented in (X)L using the [] operator. In the example below, *traveller* is the 'primary', while *drive*, *fly* and *sail* are the 'secondaries'. In the graphical depiction of this example (figure 4.18), the 'primary' is connected by an arc to the top of the rectangle, and the 'secondaries' are connected by arcs to the bottom of the rectangle.

traveller[transport] || (drive[transport] [] fly[transport] [] sail[transport])

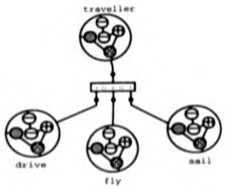


Figure 4.18: A *once-only switch combinator* joins four *components*

4.4.5 Components

A component is a unit of structure. Often, what is identifiable as a component at the specification/design stage will also be identifiable as a component at the implementation stage. Components from the *common architectural component* layer of our pyramid of architecture provide more concrete embodiments of the concepts from the lower layers of our architecture.

4.4.6 Def. CC1 Functional components

Subclassification within *functional components* is based upon Turner's "implementation functions" in [Tur88b]. Figure 4.19 shows a *functional component* graphic.

Figure 4.19: *Functional component* graphic

4.4.6.1 Def. CC2 Storage components

Informal description The important characteristics of a *storage component* are: that no transformation of data takes place between its inputs and outputs; it has a small, constant number of inputs and outputs (perhaps just one of each); and buffering is the primary characteristic. Storage components are often classified according to the scheme used to retrieve the data stored in their buffers, e.g. LIFO, FIFO, key-indexed, etc. Figure 4.20 shows a *storage component* graphic.

Figure 4.20: *Storage component* graphic

Formal representation The following LOTOS process displays the key aspects of a *storage component*.

```
(* Component class: storage component *)
process storage_comp [g] (buffer:BufferSort) : noexit :=
        g ! Input ? data:DataSort; (* get data to be stored *)
        storage_comp[g](Insert(data,buffer)) (* store the data *)
    []
        g ! Output ? retrieval_scheme:Retrieval_schemeSort ? data:DataSort
             [data eq RetrieveByScheme(retrieval_scheme,buffer)];
             (* retrieve already stored data *)
        storage_comp[g](Delete(data,buffer))
endproc (* storage_comp *)
```

No data transformation occurs between an *Input* and *Output*, and data is retrieved according to some given *retrieval_scheme*.

4.4.6.2 Def. CC3 LIFO and Def. CC4 FIFO components

Informal description The primary function of a *FIFO* or *LIFO component* is to specify a queueing discipline for events. Queues, depending on the time data spends in the queue and the emphasis placed upon this delay, may be used to realize either *storage components* or *asychronous communication components* — in figure 4.12 we choose to classify *LIFO* and *FIFO components* as *storage components*. Figure 4.21 shows *FIFO component* and *LIFO component* graphics.

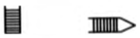


Figure 4.21: *LIFO* and *FIFO component* graphics

Formal representation The essence of a simple *FIFO component* is specified by the following LOTOS process. (LOTOS text for a *LIFO component* is similar.)

```
(* Component class: FIFO component *)
type FIFO_QType is DataType
   sorts FIFO_QSort
   opns InqueueData: DataSort, FIFO_QSort -> FIFO_QSort
         DequeueData: FIFO_QSort -> DataSort
         DiscardData: FIFO_QSort -> FIFO_QSort
         IsEmpty: FIFO_QSort -> Bool
         {} -> FIFO_QSort
   eqns
      forall q: FIFO_QSort, x: DataSort
      ofsort Bool
         IsEmpty({}) = True;
         IsEmpty(InqueueData(x,q)) = False;
      ofsort FIFO_QSort
         DiscardData(InqueueData(x,{})) = {};
         not(IsEmpty(q)) =>
            DiscardData(InqueueData(x,q)) = InqueueData(x,DiscardData(q));
      ofsort DataSort
         DequeueData(InqueueData(x,{})) = x;
         not(IsEmpty(q)) =>
            DequeueData(InqueueData(x,q)) = DequeueData(q);
endtype (* FIFO_QType *)

process FIFO_comp [g] (q:FIFO_QSort) : noexit :=
      g ! Input ? data:DataSort;
      FIFO_comp[g](InqueueData(data,q))
   []
      ([not(IsEmpty(q))] ->
      g ! Output ! DequeueData(q);
      FIFO_comp[g](DiscardData(q)))
endproc (* FIFO_comp *)
```

4.4.6.3 Def. CC5 Transformational components

Informal description The important characteristics of a transformational component are: that data is transformed from its inputs to its outputs; it has a small, constant number of inputs and outputs (perhaps just one of each); and that it implicitly carries a small amount of buffering between its inputs and outputs. Transformational components may compute data transformations themselves (by means of algorithms or look-up tables), or may solicit help to perform the transformational computation from server components. Figure 4.22 shows a *transformational component* graphic.

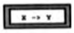


Figure 4.22: *Transformational component* graphic

Formal representation The essence of a transformational component is specified by the following LOTOS process.

```
(* Component class  transformational component *)
process trans_comp [g] : noexit :=
    g ! Input ? data DataSort; (* input data to be transformed *)
    g ! Output ! xFunction(data); (* output transformed data *)
   trans_comp[g]
endproc (* trans_comp *)
```

Data is transformed by the *xFunction* operation. The minimum amount of implicit buffering takes place between the *Input* and *Output* events.

4.4.6.4 Def. CC6 Asynchronous communication components

Informal description The important characteristics of an asynchronous communication component are: that no transformation of data takes place between its inputs and outputs; it has a large, possibly dynamically changing, number of inputs and outputs; and that it implicitly carries a small amount of buffering between its inputs and outputs. Asynchronous communication components may perform multicast functions (copying and distributing data from one input to many outputs), or interleaving functions (collecting data from many inputs and scheduling it into one output). Figure 4.23 shows an *asynchronous communication component* graphic.

Figure 4.23: *Asynchronous communication component* graphic

Formal representation An *asynchronous communication* component must be represented by at least two LOTOS events: an input event and a subsequent output event. The interval delimited by these two events realizes the asynchronicity in the communication. (A single event synchronization may be regarded as a synchronous communication — see Def. B8 and Def. CS1-CS6.)

The essence of an asynchronous communication component is specified by the following LOTOS process.

```
(* Component class: asychronous communication component *)
process async_comms_comp [g] : noexit :=
    g ! Input ? data DataSort @t1; (* input the data to be transmitted, *)
                                   (* at time t1 *)
    g ! Output ! data {xTimeSortFunction(t1)}; (* output the transmitted data, *)
                         (* communication delay is computed by xTimeSortFunction *)
                         (* '{}' and '@' are XL syntax *)
    async_comms_comp[g]
endproc (* async_coms_comp *)
```

In the XL specification '*xTimeSortFunction(t1)* minus *t1*' represents the communication delay associated with this *asynchronous communication component*.

4.4.7 Def. CC7 Performance components

The primary concern of performance components is the specification and manipulation of metrics. We subclassify performance components into components concerned with timing, probability, priority and resource metrics. Figure 4.24 shows a *performance component* graphic.

Figure 4.24: *Performance component* graphic

4.4.7.1 Def. CC8 Timing components

The primary function of a *timing component* is to specify quantitative timing constraints. Within a *timing component* we may specify that events occur only at constrained times, events occur as late/soon as possible, or we measure the duration between events, etc. See chapter 6 for a full explanation of the development and use of the quantitative timing features supported by XL.

Figure 4.25: *Timing component* graphic

In protocol design, the most common example of a *timing component* is the timeout mechanism, but in distributed system design in general, we find many instances where quantitative timing is important, e.g. time stamping of messages, re-synchronizing and guaging the error limits of local clocks, providing regular pulses to clock-tick driven components, etc. Figure 4.25 shows the graphic we use to depict a generic *timing component*. Below we list three subclasses of *timing component*.

4.4.7.2 Def. CC9 Clock components

Informal description Most distributed systems employ sets of physical clocks. Real-time distributed systems are regulated by a set of synchronized physical clocks, and non real-time systems often use physical clocks to establish causality, message ordering, etc. (see section 6.3.12). Figure 4.26 shows a *clock component* graphic.

Figure 4.26: *Clock component* graphic

Formal representation For the development of many distributed system designs we can assume the existence of a set of distributed, well synchronised, physical clocks (see section 6.3.12). Hence, with no need to describe how time tick information is realized, we concentrate on 'declaratively' specifying the timing constraints which system components must satisfy. XL provides the luxury of an in built time-keeping mechanism, thus rescuing the specifier from the time-consuming task of building in a time-keeping, time-distribution mechanism. XL's time features allow the specifier to 'declaratively' describe the quantitative timing constraints for the systems components. Under this scenario, we might introduce the *clock component* graphic into the graphical description of the system under design, just to make explicit that some of the system's components are time dependent. And we might link, by arcs, the time dependent *component* graphics to the *clock component* graphic. However, under this scenario, there will be no need to associate the *clock component* with any XL text, since the clock mechanism is implicit in XL.

However, sometimes the design of a distributed system may involve the design of time-keeping and time-distributing mechanisms themselves. In this scenario, these mechanisms are an integral part of the problem to be solved and should be given explicit representations in the XL specification. In such a scenario, *clock component* graphics (and the arc connections to the other *components* in the system) will be associated with XL text describing the construction of the clocks, how they distribute time information, how they synchronise, etc. The XL text describing the construction of a *clock component* may use XL's built-in time mechanism like a physical clock uses a vibrating quartz crystal, as a means of sensing the passage of time. However, the actual time kept by the *clock component* (in this scenario) will not only be a function of XL's built-in time, but also a function of the times registering on the other supposedly synchronous physical clocks in the system. (We would expect that the specification of such clocks might embody algorithms for maintaining synchronisation between distributed clocks to within calculatable error limits, such as described by [Lam78].)

(The advice from the previous paragraph, that "mechanisms that play an integral part in the problem to be solved should be given explicit representation in an XL description" can be put into a more general discussion: It is always the case that it is much easier to state that "there should be a mechanism to..." than to describe the mechanism itself. The expressiveness of XL is such that it may be especially easy to to describe certain classes of system at an abstract level (in the problem domain) as compared to their less abstract descriptions (in the solution domain). These particular classes of system usually involve synchronisation and concurrency. Their abstract descriptions are easy to formulate because they employ implicit features of the XL model, such as synchronisation, whereas their less abstract descriptions must explicitly describe a synchronisation mechanism.

When using XL in the development of a system, we must be careful not to overlook problems because of their implicit treatment by XL. Consider a distributed system where the total ordering of events is difficult to establish. If we rely on the XL features of implicit synchronisation and event ordering we may miss the crux of the problem (establishing a total event ordering on the basis of asynchronous communication) until later in the design process. We must take care that the XL description really reflects the essence the problem.)

4.4.7.3 Def. CC10 Timeout components

Informal description In protocol specifications, in particular, timeout behaviour accounts for most of the quantitative time dependent aspects of the behaviour of the systems. The primary characteristic of a *timeout component* is that it specifies the time at which some 'exception behaviour' is to be taken, in case a reply has not been received. Figure 4.27 shows a *timeout component* graphic.

Figure 4.27: *Timeout component* graphic

Formal representation The following XL process displays the key aspects of a *timeout component*.

```
(* Component class  timeout component *)
process timeout_comp [g]   noexit :=
    g ! Request ? data DataSort @t1; (* send request *)
    (
        g ! Confirm ? data DataSort {setLE(t1+timeout_period)} ASAP;
                        (* confirm occurs within the timeout period *)
        timeout_comp[g]
    []
```

```
        i {setEQ(t1+timeout.period+1)}; (* timeout occurs *)
        take_exception_behaviour_for_timeout[g]
    )
endproc (* timeout_comp *)
```

See sections 6.2 and 6.7 for further discussions on how to specify timeouts in XL.

4.4.7.4 Def. CC11 Stopwatch components

Informal description The primary characteristic of a *stopwatch component* is that it measures the duration between event occurrences. Such information may be used to monitor system performance issues such as processor utilisation, or used in the compilation of logs recording activity times, etc. Figure 4.28 shows a *stopwatch component* graphic.

Figure 4.28: *Stopwatch component* graphic

Formal representation The following XL process displays the key aspects of a *stopwatch component*.

```
(* Component class: stopwatch component *)
process swatch_comp [g] : noexit :=
    g!a @t1; (* note time of 'start' event *)
    g!b @t2; (* note time of 'finish' event *)
    (* duration between 'start' and 'finish' events is t2-t1 (of type TimeSort) *)
    swatch_comp[g]
endproc (* swatch_comp *)
```

4.4.7.5 Def. CC12 Probability components

Informal description The primary purpose of a *probability component* is to specify probability or statistical constraints. Examples of such constraints include the specification of the probability of an event occurrence, or the description of the frequency distribution over a range of possible event occurrences. See chapter 7 for a full explanation of the development and use of the probability features supported by XL. Figure 4.29 shows a *probability component* graphic.

Figure 4.29: *Probability component* graphic

Formal representation The following XL process displays the key aspects of a *probability component.*

```
(* Component class: probability component
   The probability distribution
   between the three events is 5:3:2. *)
process prob_comp [g] : noexit :=
   g!a; stop [=0.5] (g!b; stop [=0.6] g!c; stop)
endproc (* prob_comp *)
```

4.4.7.6 Def. CC13 Priority components

Informal description The primary purpose of a *priority component* is to specify priority constraints. Priority constraints specify the relative priorities between events of the same priority class. See section 8.1 for a full explanation of the development and use of the priority features supported by XL. Figure 4.30 shows a *priority component* graphic.

Figure 4.30: *Priority component* graphic

Formal representation The following XL process displays the key aspects of a *priority component.*

```
(* Component class: priority component
   The priorities are 4:1:7 *)
process priority_comp [g,f] : noexit :=
      g!a #(class1,4) (* middle priority *); stop
   [] g!b #(class1,1) (* lowest priority *); stop
   [] f #(class1,7) (* highest priority *); stop
endproc (* priority_comp *)
```

4.4.7.7 Def. CC14 Resource management components

Figure 4.31: *Resource management component* graphic

Informal description The primary purpose of a *resource management component* is to specify how many instances of a resource exist. The order in which resources are allocated with respect to the order and priority of the requests for a resource will normally be handled by *priority components* and *asynchronous communication components* connected to the *resource management component*.

Formal representation The following example of a *resource management component* specifies the number of concurrent data units which can be transmitted by a set of *asynchronous communication components* (described in Def. CC6). The *resource management component* (*res1_comp* instantiated with $n = 0$) launches a static finite number *max* of instances of the *asynchronous communication component*. Access to a resource instance is controlled by the resource instance process itself (*async_comms_comp*) which permits one client at a time to synchronise with it.

```
(* Component class: resource management component *)
process res1_comp [g] (n:Nat) : noexit :=
        ([n lt max] -> (* launch another resource instance *)
            async_comms_comp[g] ||| res1_comp[g](Succ(n)))
    []
        ([n ge max] -> (* max resource instances launched, so stop *)
            stop)
endproc (* res1_comp *)
```

By classification, the *res1_comp component* should be graphically depicted as a *resource management component* (figure 4.31). However, this obscures the primary purpose of the *component* which is asynchronous communication. Alternately, depicting the *res1_comp* as an *asynchronous communication component* graphic (figure 4.23) does indicate the primary purpose of the *component* but it still masks the important resource management aspect of the *component*. Masking the resource management aspect of *res1_comp* (by the asynchronous communication aspect) tends to violate our commitment to the description principle of *separation of concerns*. The crux of the problem lies in the way in which the *component* is structured, with the *resource management component* completely encompassing the resource (i.e. the set of *asynchronous communication components*). Our second *resource management component* example (*res2_comp* below) adheres to the *separation of concerns* principle by structuring the system such that the *resource management component* and the resource *component* are distinct, but synchronising *components* (see figure 4.32, left).

The second example below of a *resource management component* specifies the number of data units which can be concurrently stored and retrieved from a storage resource formed by a set of *storage components* (described in Def. CC2). Unlike our first *resource management component* example, the *resource management component* in this example stays active throughout the lifetime of the resource that it manages.

```
(* Component class: resource management component *)
process res2_comp [g] (n:Integer) : noexit :=
        ([n lt max] -> (* storage available, allow data to be stored *)
            g ! Input ? any DataSort;
            res2_comp[g](n+1))
    []
```

are allocated with respect to the order and priority of the requests for a resource will normally be handled by *priority components* and *asynchronous communication components* connected to the *resource management component*.

Formal representation The following example of a *resource management component* specifies the number of concurrent data units which can be transmitted by a set of *asynchronous communication components* (described in Def. CC0). The *resource management component* (*res1_comp* instantiated with $n = 0$) launches a static finite number *max* of instances of the *asynchronous communication component*. Access to a resource instance is controlled by the resource instance process itself (*async_comms_comp*) which permits one client at a time to synchronise with it.

```
(* Component class: resource management component *)
process res1_comp [g] (n:Nat) : noexit :=
        ([n lt max] -> (* launch another resource instance *)
            async_comms_comp[g] ||| res1_comp[g](Succ(n)))
    []
        ([n ge max] -> (* max resource instances launched, so stop *)
            stop)
endproc (* res1_comp *)
```

By classification, the *res1_comp component* should be graphically depicted as a *resource management component* (figure 4.31). However, this obscures the primary purpose of the *component* which is asynchronous communication. Alternately, depicting the *res1_comp* as an *asynchronous communication component* graphic (figure 4.23) does indicate the primary purpose of the *component* but it still masks the important resource management aspect of the *component*. Masking the resource management aspect of *res1_comp* (by the asynchronous communication aspect) tends to violate our commitment to the description principle of *separation of concerns*. The crux of the problem lies in the way in which the *component* is structured, with the *resource management component* completely encompassing the resource (i.e. the set of *asynchronous communication components*). Our second *resource management component* example (*res2_comp* below) adheres to the *separation of concerns* principle by structuring the system such that the *resource management component* and the resource *component* are distinct, but synchronising *components* (see figure 4.32, left).

The second example below of a *resource management component* specifies the number of data units which can be concurrently stored and retrieved from a storage resource formed by a set of *storage components* (described in Def. CC2). Unlike our first *resource management component* example, the *resource management component* in this example stays active throughout the lifetime of the resource that it manages.

```
(* Component class: resource management component *)
process res2_comp [g] (n:Integer) : noexit :=
        ([n lt max] -> (* storage available, allow data to be stored *)
            g ! Input ? any DataSort;
            res2_comp[g](n+1))
    []
```

```
      g ! Output ? any DataSort,
      res2_comp[g](n-1)
endproc (* res2_comp *)
```

This *resource management component* must be synchronised with the *storage component* which it manages, i.e.:

res2_comp[g](0) || storage_comp[g](emptyStore)

It is now sensible to depict the combination of the *res2_comp* and the *storage_comp* (figure 4.32, right) as a *storage component* (figure 4.32, left), thus indicating its primary purpose. Also, the distinct separation of the resource management aspects from the resource aspects, in this second example, allows us to sensibly decompose the *component* as shown in figure 4.32.

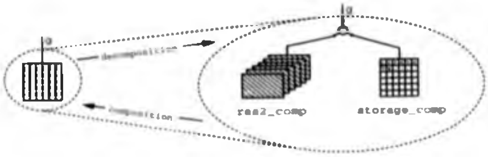


Figure 4.32: A (de)composition of a *resource management component* and its resource *component*

4.4.8 Miscellaneous structuring components

4.4.8.1 Def. CC15 Untyped components

Informal description An *untyped component* is used either to represent a *component* with no dominant characteristic, or as a 'placeholder' in the early stages of development. When used to represent a *component* with no dominant characteristic, decomposition of this *component* may reveal a number of *sub-components* each with an identifiable *component* class. When used as a 'placeholder' *component*, it is expected that the *untyped component* will be replaced by a specific class of *component* at a later stage of development. Figure 4.33 shows an *untyped component* graphic.

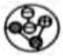


Figure 4.33: An *untyped component* graphic

4.4.8.2 Def. CC16 Client-rôle interface component

A *client-rôle interface component* is a *sub-component* of a parent *component* (see Def. CC18 for an example). The *client role interface component* realizes an interface of the parent *component* through which the parent *component* solicits the service offered by other *components*. Figure 4.34 shows a *client-rôle interface component* graphic.

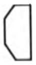


Figure 4.34: A *client-rôle interface component* graphic

4.4.8.3 Def. CC17 Server-rôle interface component

A *server-rôle interface component* is a *sub-component* of a parent *component* (see Def. CC18 for an example). The *server rôle interface component* realizes an interface of the parent *component* through which the parent *component* offers a service to other *components*. Figure 4.35 shows a *server-rôle interface component* graphic.

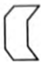


Figure 4.35: A *server-rôle interface component* graphic

4.4.8.4 Def. CC18 Client/server components

A *client component* solicits the help of *server components* to perform functions on its behalf. The composition of a *client component* will include at least one *client-rôle interface component*.

A *server component* performs functions on behalf of *client components*. The composition of a *server component* will include at least one *server-rôle interface component*.

A *client/server component* acts in both client and server rôles. The composition of a *client/server component* will include at least one *client-rôle interface component* and at least one *server-rôle interface component*.

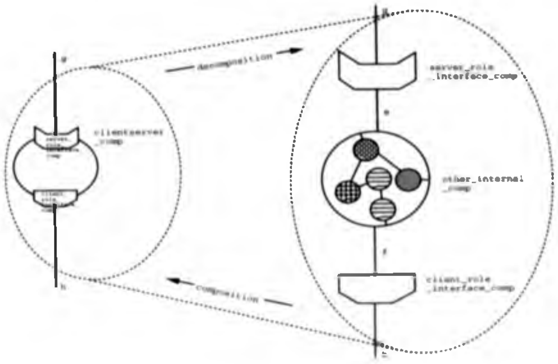


Figure 4.36: A *client/server component* graphic

Normally we do not depict a graphic representing a *sub-component* at the same level of (de)composition as the graphic depicting the parent *component*. However, in the graphics depicting *client/server components* we make a concession for the sake of descriptive power. Graphics depicting *client/server components* are presented as ovals with *client-* and *server-rôle interface component* graphics imposed (e.g. figure 4.36, left).[15] Also, we allow each *client/server-rôle interface component* graphic, within its parent *client/server component* graphic, to be given a label.

Figure 4.36 shows the graphic for the *client/server-component* whose XL description follows.

```
(* Component class  client/server component
   3 sub components  a client-role interface component,
   a server-role interface component, and some other component  *)
process clientserver_comp [g,h]   noexit :=
   hide e,f in (
             server_role_interface_comp[g,e])
        |[e]| other_internal_comp[e,f]
        |[f]| client_role_interface_comp[f,h]
   )
```

[15] This breaks with the convention used in the rest of this chapter because *client/server-rôle interface components* are *sub-components* of *client/server components*, and hence *client/server-rôle interface component* graphics should only be viewable when *client/server component* graphics are 'exploded' to reveal the graphics of their *sub-components*.

endproc (* clientserver_comp *)

4.4.8.5 Def. CC19 Protocol

This is a set of rules which govern how two or more *components* communicate.

4.4.8.6 Def. CC20 Service

This denotes the behaviour offered by a set of *components* (acting in a server rôle) at a set of interfaces.

4.4.9 Specific architectural components

These are architectural elements which are used for specific distributed systems problems. The *specific architectural components* for one problem area succinctly describe architectural elements of that problem, but are too specialized to be of use outside the particular problem domain.

Specific architectural components will often be built using customised *common architectural components*. Our case-study in chapter 5 describes how a set of *specific architectural components* have been built from the *common architectural components* for the particular problem of formalising the CIM-OSA IIS (Computer Integrated Manufacturing – Open Systems Architecture Integrating Infrastructure).

4.4.10 Discussion

The architecture framework presented in this chapter is not definitive. This deposition of architectural concepts should be regarded as an *indication* of the ingredients and structure in an architecture for distributed systems. Given this reservation, we have found the compilation and suggested formalisation of a framework of architectural concepts a useful base for developing distributed systems, such as CIM-OSA (chapter 5).

4.4.10.1 Language assessment with respect to representing architectural elements

Simplicity and directness In section 4.4.1 we saw how all the *basic architectural ingredients* are directly mappable to similar XL concepts. Section 4.4.2 showed us that the *architectural tools and structuring concepts* have equivalences in the XL world; and sections 4.4.3 to 4.4.9 suggested a categorisation of *common* (higher order) *architectural components*. Exercises using these mappings certainly indicate that XL meets the simplicity and directness of expression criteria, which we suggest as a heuristic for a good formal language in which to represent architectural concepts.

Wide spectrum The examples in this chapter and in chapter 5 show that XL is capable of representing concepts from all branches in our hierarchical infrastructure of architectural concepts ("horizontal coverage"). Also, [Pir91] explains how LOTOS is suitable for describing (partial) designs at arbitrary levels of abstraction throughout the development cycle ("vertical coverage"). Hence, XL can be called a "wide spectrum" language.

Compositional reasoning To manage complexity, systems are often described as compositions of smaller subsystems. An obvious requirement for XL is that it too should support some kind of compositional specification and reasoning in accordance with the compositional structure of the system it is used to describe. Examples in this chapter and in chapter 5 demonstrate how XL supports a compositional approach. Compositional reasoning about XL specifications is aided by theoretical tools such as equivalences, congruences, formal transformations, etc. (see section 7.4 and appendix G).

Quantitative time, probability and priority In our infra-structure of architecture, we have not only included elements for describing functional concerns, but also elements for describing performance concerns. We believe that performance concerns are often as important as and inseparable from functional concerns. In recognition of this importance, we have defined a class of *performance components* (section 4.4.7).

Although functional elements can be reasonably directly represented in LOTOS, LOTOS proves cumbersome for representing performance elements (especially quantitative time, probability and priority concerns). Therefore we have used the 'performance specification' features supported by XL developed in chapters 6, 7 and 8, in representing performance elements.

4.4.10.2 Alternative mappings of interest

This chapter has provided generic XL representations of architectural concepts. Alternative representations exist. For example, imagine if our concern lay more with expressing the object-oriented aspect of our architecture in a non-procedural, declarative algebra. We might use the ACT ONE part of LOTOS, e.g. [ROS89b], where objects, messages and data are mapped to sorts and operations. The type concept is realized by parameterised specification. Iterated actualization is used to support inheritance and type/subtype relations. The encapsulating properties of ACT ONE types are used to realize the opaqueness properties of objects. [Gib93] defines another way of using ACT ONE to describe object oriented systems.

4.5 Summary

This chapter began with the premiss that the design of distributed systems ought to be architecture-driven, rather than description language driven. Architecture-driven methods possess the advantage that they embody domain knowledge — know-how built up from a previous history of solutions, and organized into an infra-structure of concepts, ingredients, template components, etc. The disadvantage of architecture-driven methods is their lack of generality. (An architecture for building distributed computing systems is of little use for designing GUIs, for example.)

Accepting the sensibility of architecture-driven methods, we proceeded to build our own infra-structure for architecture of the specification of distributed systems. We looked at some fundamental ideas on the nature of description to provide a firm basis for our architecture. That established we built a pyramid of architecture, beginning with the most simple and common elements first. Architectural elements were given suggested XL representations, providing an algebraic perspective on the architecture.

Near the apex of our pyramid of architecture, we reached what we call the *common architectural components*. This set of *components* included *performance components*, as well as *functional components*, in recognition of the importance of performance specification in systems design. We gave these *components* XL templates and graphical representations, and recommended that they be customised and used in the composition of *specific architectural components* for specific distributed systems problems.

The next chapter (chapter 5) implements this recommendation, taking us from theory to practice. Chapter 5 substantiates the work of this chapter, showing how our infra-structure of architecture has served, in an industrially sponsored project, as the basis for formalising the CIM-OSA IIS distributed system.

Chapter 5

Case-study: the CIM-OSA IIS

An important area in LOTOS research is the application of LOTOS to domains other than OSI. For this thesis, the Esprit CIM-OSA project (Computer Integrated Manufacturing — Open Systems Architecture) [CIM90d, CIM90a, CIM89c] provides a challenging industrial domain for the use of LOTOS. The author has been involved in CIM-OSA attempts to develop a formal model of parts of the CIM-OSA architecture. As a case-study, this chapter illustrates how XL, together with chapter 4's infra-structure of architecture can be used to model and formalise a part of the CIM-OSA reference architecture known as the IIS (Integrating Infrastructure). Since both functional and performance specification play important rôles in CIM systems, CIM-OSA is a suitable case-study for testing the descriptive power of XL.

We find architecture-driven specification, the conviction of chapter 4, to be useful not only in the initial stages of specification but also in the later stages. This is due to the close relationship between the problem architecture and specification architecture. This closeness helps guide the specifier during the initial stages, and helps the specifier navigate around and understand the solution specification in later stages.

Regarding the use of LOTOS (and XL), we find that this formalism provides a sound framework for reasoning about development. In particular, its rigour promotes early problem identification.

5.1 Introduction

This section provides an introduction to the CIM-OSA IIS, in preparation for the development of its specification in the following sections. This section also briefly discusses the benefits created from the marriage of CIM-OSA and XL.

5.1.1 Introduction to the CIM-OSA IIS

In section 2.3.5 we provided a brief introduction to the history, objectives and structure of the CIM-OSA project. Here we elaborate on the part of the CIM-OSA reference architecture which concerns us: the **Integrating Infrastructure (IIS)**.

Section 2.3.5 placed the IIS in its CIM-OSA context. We identified that the IIS is the part of CIM-OSA which is responsible for providing a set of services common to the needs of most CIM systems. We can think of the IIS as an information technology platform onto which any particular CIM-OSA system can be built. This rôle has earned the IIS the title of the "CIM-OSA Operating System"[Bee89].

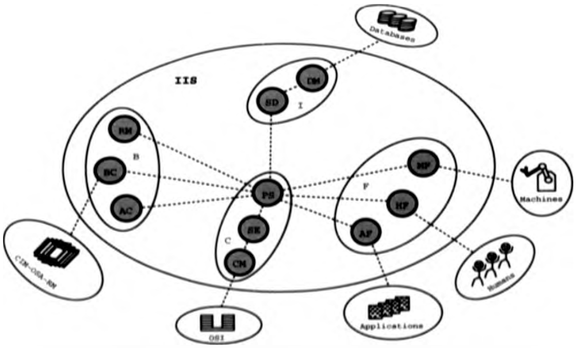


Figure 5.1: Structural composition of the IIS

Figure 5.1 shows the coarse structural composition of the IIS. From figure 5.1 we see that the IIS is a composition of 4 major entities: the **Business Complex (B)**, the

Front-End Complex (F), the **Information Complex** (I), and the **Communications Complex** (C).

5.1.1.1 Business complex (B)

The Business Complex consists of the **Business Process Control Service** (BP), the **Activity Control Service** (AC) and the **Resource Management Service** (RM).

BP executes 'business programs'[1]. These programs are susceptible to modification, dependent upon the relatively unstable short term goals of the enterprise. The execution of 'business programs' involves BP managing the sequencing and synchronization between the more stable 'business activities'[2]. BP is also responsible for managing the release and integration of new 'business programs' and 'business activities'.

AC performs a task similar to BP, but for 'business activities'. Executing 'business activities' involves the management of 'functional operations'[3].

RM provides system-wide management of the resources used in the execution of 'business programs' and 'business activities'.

5.1.1.2 Information complex (I)

The Information Complex consists of the **System-Wide Data Service** (SD) and the **Data Management Service** (DM).

SD presents a coherent means of storing, retrieving and managing schema conversions. Clients may remain ignorant of actual data distribution and actual storage schema. SD manages the integration of local DBMSs (Data Base Management Systems) and allows clients to request data in schema specified by them. SD is also responsible for access authentication and data integrity.

Each DM performs the rôle of interpreter between the particular DBMS and the SD, so facilitating the integration of vendor-specific DBMSs into a CIM OSA system.

5.1.1.3 Front-end complex (F)

The Front-End Complex consists of the **Human Front-End Service** (HF), the **Machine Front-End Service** (MF) and the **Application Front-End Service** (AF).

These services present application programs, humans and machines (i.e. the functional units which finally perform the enterprise functions) to the rest of the IIS in a homogeneous way, and vice versa.

5.1.1.4 Communications complex (C)

The Communications Complex consists of the **Protocol Support Service** (PS), the **System-Wide Exchange** (SE) and the **Communications Management Service**

[1]**Implemented Business Processes**, in CIM-OSA terminology.

[2]**Implemented Business Activities**, in CIM-OSA terminology.

[3]**Implemented Functional Operations**, in CIM-OSA terminology.

(CM).

PS[4] is responsible for mapping **access-protocol** and **agent-protocol** communications, between the B, I and F, onto suitable SE communication services. PS provides a certain degree of distributed communications transparency to its users, by handling communication failures, retry schemes, addressing information, etc.

SE provides a system-wide homogeneous platform for data communication. It offers the basic level of service needed to support the demands for communication made on it from PS in its support of agent-protocols and access-protocols.

CM acts as an intermediary between the supporting OSI or vendor-specific communications services and the rest of the CIM-OSA system.

5.1.2 Justification (and related work)

5.1.2.1 The benefits from CIM-OSA for (Extended) LOTOS

A number of authors (e.g. [VSvSB90, vS90, Tur87, ISO93a]) have already documented ideas and strategies for formalising system/architecture design and development using LOTOS. The two reference architectures featured in this work are OSI and ODP.

OSI describes communications systems using a symmetric, layered architecture. In contrast, CIM-OSA specifies systems which cannot be described solely in terms of hierarchical strata due to the asymmetric composition of the IIS. The IIS consists of a number of heterogeneous components whose communications form a complex web of interaction and dependency.

ODP forms a very general reference framework for distributed systems description. CIM-OSA is interesting because it represents a much more applied and specialized reference architecture. CIM-OSA implicitly uses many of the ODP-like architectural concepts (which we have elaborated and suggested formal representations for in chapter 4) to define architectural concepts specific to CIM systems.

The application of LOTOS to the CIM-OSA IIS provides an insight into the advantages of building a distributed system upon a pre-defined framework of formalised architectural concepts (as defined in chapter 4).

Both functional and performance specification play important rôles in CIM systems. This makes the CIM-OSA IIS an excellent testing ground for assessing the performance specification features (quantitative timing, probability and priority) of XL developed in chapters 6, 7 and 8 of this thesis.

We have had the opportunity to observe the effects which LOTOS has had on the CIM-OSA project, and how the application of LOTOS has gradually developed. The developers of CIM-OSA come from a wide variety of technical backgrounds (e.g. electrical engineering, management, automotive and aerospace manufacturing, software engineering). It has been interesting to see how people from such differing 'cultures' embrace the use of a formal description technique.

[4]This service did not originally exist in the CIM-OSA architecture, but was identified as necessary during the process of formalising the IIS by the author, see [McC90a].

5.1.2.2 The benefits of (Extended) LOTOS for CIM-OSA

The CIM-OSA project has captured its reference architecture in a volume of documents known as the CIM-OSA **Formal Reference Base** (FRB) (e.g. [CIM89b, CIM90b, CIM89a, CIM90c]). The FRB provides systematic but *informal* (English language text with supporting diagrams) descriptions of the IIS. Certain aspects of these descriptions are incomplete (at the specification level), with structural, functional and informational elements missing. Ambiguity is another problem found in FRB descriptions. This is a result of both the ambiguity inherent in natural language prose, and the absence of definitive descriptions of some of the architectural concepts used within CIM-OSA. Also, inconsistences occur in the descriptions of IIS subsystems and their interworkings.[5]

Once the FRB was established, the need to introduce some kind of formalism to all areas of the project (IIS and other aspects of CIM-OSA's reference architecture) became obvious, and LOTOS was chosen to this end (to formalize IIS descriptions). The development of LOTOS descriptions of IIS elements has helped identify the above mentioned problems of incompleteness and inconsistency. The creation of LOTOS descriptions has forced decision making processes which solve ambiguities and other problems. This LOTOS-supported development process has made designers conscious of issues such as: levels of abstraction; the identification of structural, functional and informational elements within the CIM-OSA architecture which are important at the specification level; and what constitutes a good specification level design and why. Moreover, since the IIS is part of a "reference architecture" it is even more desirable that its description have all the properties of a formal representation.

Also, since CIM-OSA is a reference architecture still in its infancy, we believe that formalising has helped catch many design flaws and oversights at, perhaps, an earlier stage than normal — an advantage predicted by those acclaiming "early prototyping", e.g. [AJ89].

5.1.3 The choice of LOTOS

Having taken the decision to employ some kind of formal technique in the development of the IIS, the project set up a task force [CO89] to investigate existing formal languages and recommend the most suitable. A short list included the three FDTs LOTOS, SDL and Estelle. Detailed, expert comparisons of these three FDTs can be found in the literature (e.g. [CO89]). We briefly present some of the criteria used by the investigatory task force, as it gives an indication of what the project hoped to gain from an FDT, and because it portrays what one potential consumer of FDT technology saw as the relative benefits.

One concern of the FDT task force was with the *functional coverage* of FDTs. The task force examined concurrency, sequentiality, data, system testing and real-time aspects of the FDTs. The other not (yet) standardized formal languages VDM and Z, based on predicate calculus, were rejected because of their lack of built-in facilities for expressing

[5] We would like to emphasise that such problems are by no means unique to the CIM-OSA project but are characteristics of informal descriptions in general.

concurrency. The task force concluded LOTOS to be the most powerful with respect to concurrency aspects, with its interleaving, enabling and disabling features, and because of its synchronous basis[6]. None of the three FDTs met the real-time criteria. It was felt that the ability to easily express performance constraints forms an important consideration, especially in view of the time- (and safety-) critical nature of many manufacturing operations. The task force concluded that, if necessary, one of the approaches for a pseudo real-time could be adopted.

Note: This CIM-OSA task force conclusion is realized in this thesis: extensions to LOTOS for performance specification are developed in chapters 6, 7 and 8.

The FDT task force studied *formal definition* concerns. These included syntax and semantics, for which LOTOS scored highest; analyzability, for which LOTOS faired well with its theories for equivalences, transformations, etc.; computability, for which LOTOS passed because it could support the required level of prototyping; implementation independence, for which LOTOS was considered the most abstract; and international standardization, which all three FDTs have achieved.

Under the concern of *human orientation*, the LOTOS syntax was thought to be esoteric, and the lack of a standard graphical representation a drawback.

Note: Chapter 4 advocates building and reasoning about distributed systems in terms of architectural components, rather than in terms of LOTOS. Architectural concepts tend to be much closer to the problem domain, thus more 'designer friendly' than XL concepts. Also, chapter 4 suggests graphical representations which can be used in conjunction with (XL based) architectural descriptions to aid the readability of system designs.

The learning curve for LOTOS, compared to the less expressively flexible but easier to learn SDL and Estelle, might have adversely affected project time scales — a number of project members would have to be trained in LOTOS to a level of sufficient expertise, if the initial formalisation phase were to be extended.

LOTOS takes the lead in *expressive power* with its ability to express non-determinism, its mixture of declarative (ACT ONE) and procedural (process algebra) styles, and with the ease of expression it affords to complex concepts such as concurrency and multi-way synchronization.

Other concerns such as *tools, structuring and reusability* were also considered.

5.1.3.1 Substituting LOTOS for XL

The CIM-OSA project chose LOTOS to formalise the IIS, but for the purposes of this thesis[7] we have elaborated CIM-OSA LOTOS specifications to XL specifications.

[6] Actually, there was some fear that the synchronous basis of LOTOS would be inappropriate because of the essentially asynchronous communications in real CIM systems. Of course, it was pointed out that asynchrony can easily be modelled by synchrony, though not vice versa. Also, synchrony can be used to mask at the specification level any implementation dependent asynchronous based realisations.

[7] which include assessing XL's ability to naturally and directly express performance concerns

5.2 Our approach to formalising the IIS

In section 5.1.2.2 we answered the question *why* formalise with a list of somewhat intangible benefits, such as unambiguity, completeness, gaining experience, etc. Although of primary importance, these results are side-effects of our actual tangible products: formal IIS specifications. Our strategy for developing the IIS specifications was to:

- study the FRB and identify *IIS architectural components*
- design *IIS architectural components* in terms of the *common architectural components* (sections 4.4.3 to 4.4.8)
- write XL text to customize specific *IIS architectural components*
- provide a development map.

The FRB mentions a number of *IIS architectural components* such as **service**, **agent-protocol**, **access-protocol**, **interface**, **service-agent**, **client**, **system-wide service**, **timeout**, **asynchronous communication**, etc. These *components* are used to describe IIS systems, but many are not given any founding definitions, or are given unsatisfactory definitions.

In order to substantiate FRB definitions of the IIS, we describe *IIS architectural components* in terms of the *common architectural components* defined in sections 4.4.3 to 4.4.8. *IIS architectural components* are specific to the CIM-OSA reference architecture, and so we consider that the *IIS architectural components* lie in the topmost layer (the *specific architectural components*, section 4.4.9) of the pyramid of architecture (figure 4.6) that we defined in chapter 4.

Structuring the IIS in terms of *common architectural components* provides a skeleton description of the IIS. The next task is to 'flesh out' the skeleton description by taking each particular *component* instance and customizing its XL description to reflect its unique contribution to the IIS system.

Our final task is to provide guidelines which assist developers to interpret the suite of XL specifications and use these in further development work (e.g. design refinements, implementation, conformance testing, etc.).

In the following sections we take a look at these tasks in more detail. We point out some of the questions raised and answered by the formalising process.

5.3 A skeleton of architectural components

5.3.1 The gross architecture of the IIS

Figure 5.2 places the IIS in context. The *IIS*, the *IIS_concerned_world*, and the *rest_of_the_world* are represented by three *components* forming a closed system. The *IIS* and the *IIS_concerned_world* interact through an interface labelled *IISgates*. Here, *IISgates* is not the name of actual (X)L gates but rather a reference to a set of gates.

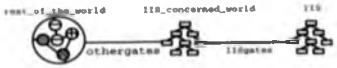


Figure 5.2: The IIS in context

```
(* Component class: untyped component *)
(* Comments: Context for the IIS  *)
specification IIS_context : noexit

behaviour
   hide IISgates, othergates in (
          IIS[IISgates]
      |[IISgates]|
          IIS_concerned_world[IISgates,othergates]
      |[othergates]|
          rest_of_the_world[othergates]
   )
   where

endspec (* IIS_context *)
```

In figure 5.3 we decompose the *IIS* and the *IIS_concerned_world components* of figure 5.2. (In this decomposition we ignore the interface between the *IIS_concerned_world* and the *rest_of_the_world*.) From figure 5.3 and the accompanying (X)L description we can see that:

- *IISgates* represents a union of the *Cgates*, *Dgates*, *Hgates*, *Agates*, *Mgates* and *OSIgates*.
- The *Bgates*, *Igates* and *Fgates* are wholly contained within (internal to) the *IIS component*.
- The *B*, *I* and *F components* communicate with one another only via the *PS component* (and ultimately via the *OSI component*). In fact, all intra-IIS communications are ultimately routed via *PS* to *OSI*.
- *B*, *I* and *F* each use their own subset of the *IISgates* to interface with the *components* in the *IIS_concerned_world*, with which they directly interact. Thus, for example, the *I component* (Information Services Complex) interfaces with vendor specific database applications via *Dgates*, and presents this variety of database applications as a consistent database system to the rest of the *components* in the *IIS* via *Igates*.

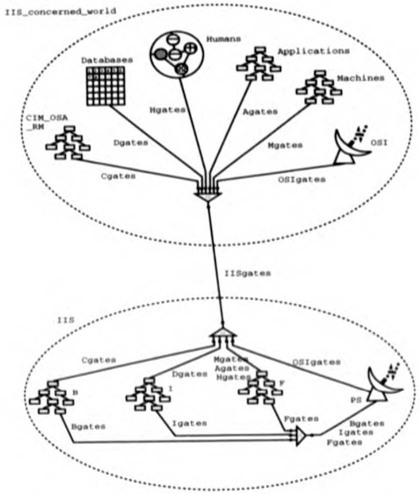


Figure 5.3: Decomposition of the IIS and the IIS_concerned_world

```
(* Component class: functional component *)
(* Comments: The IIS. *)
process IIS [Cgates,Dgates,Mgates,Agates,Hgates,OSIgates] : noexit :=
    hide Bgates,Igates,Fgates in (
        (
            B[Cgates,Bgates]
        |||
            I[Dgates,Igates]
        |||
            F[Mgates,Agates,Hgates,Fgates]
        )
    |[Bgates,Igates,Fgates]|
        PS[OSIgates,Bgates,Igates,Fgates]
    )
where
    ...
endproc (* IIS *)
```

In the following subsections we move away from a global view of the IIS to define some generic IIS concepts.

5.3.2 The IIS client-server model

CIM-OSA uses the "client-server model" as the basis of communication within the IIS. Figure 5.4 provides an overview on the client-server template. This template has several instances within the IIS, and so we use "X" as a placeholder for any legitimate IIS service instance. Creation of this client-server template simply involves annotating, with CIM OSA terminology, the Def. CC18 *component* which we have already defined in section 4.4.8.4.

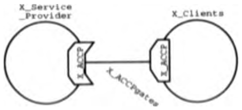


Figure 5.4: The IIS Client-Server Model

X Clients represents a set of **X Service Users**. The **X Service Provider** (X SP) is responsible for providing an **X Service** to the X Service Users. X Service Users communicate with the X Service Provider using the **X ACCP** (X Access Protocol).

We classify X Clients as *client components* and X Service Providers as a *server components* (Def. CC18). The X Clients *client component* specification will embody a *client-role interface component* Def. CC16 defining "user oriented" aspects of the X ACCP. The X Service Providers *server component* specification will embody a *server-rôle interface component* Def. CC17 defining "provider-oriented" aspects of the X ACCP.

Note: Often the FRB describes IIS *client/server components* only in terms of, what section 4.4.8.4 calls *client/server-rôle interface components*. The set of *client/server-rôle interface components* for any one *client/server component* may not *fully* define the behaviour of the *client/server component*, but rather *sufficiently* define the behaviour of the *component* for the purposes of CIM-OSA.

Thus for, say, the client server model of figure 5.4, the FRB may define the X Clients *component* only to the extent defined by the "user-oriented" X ACCP specification.

5.3.3 A system-wide service

IIS X-Service Providers provide what is known as a **system-wide** service [CIM90e] to their X_Clients. X_Clients may remain ignorant of the actual distribution of the X_Service_Provider. CIM-OSA assumes X_Service_Providers are strongly distributed systems, realized by a distributed set of inter-communicating, **X_Service_Agents**. X_Service_Agents communicate using an **X_AGEP** (X_Agent_Protocol).

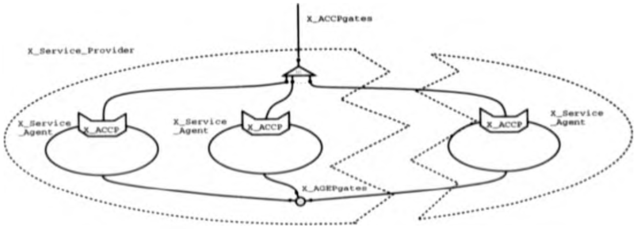


Figure 5.5: A decomposition of an X_Service_Provider

Each X_Service_Agent has two interfaces: an interface to *X_ACCPgates* and an interface to *X_AGEPgates*. From figure 5.5 we see that the *X_ACCPgates* interfaces of the X_Service_Agents, are multiplexed to form the *X_ACCPgates* interface of the X_Service_Provider.

The X_Service_Agents communicate via *X_AGEPgates* using the X_AGEP protocol. The situation portrayed in figure 5.5 is an abstraction. At a less abstract level, the X_Service_Agents do not intercommunicate directly with each other, as shown, but instead their intercommunications are routed via PS (section 5.3.5). We may specify an X_Service_Agent as the combination of the interfaces to *X_ACCPgates* and *X_ACCPgates*. OSI would call the XL specification of an X_Service_Agent a "protocol specification of service X".

5.3.4 Two important IIS structural organizations

While structuring the IIS in terms of *components* we encountered two important structural organizations:

A component providing and wholly containing a service. This is a situation where a single *component* is responsible for offering and providing the realisation, wholly contained within the *component*, of a service. To perform the service the *component* need not solicit help from any other objects.

A component offering but not wholly containing a service. This describes a situation where a *component* may offer a service but does not, alone, provide the complete realisation of that service. This normally occurs as a collection of *components* interacting with one another in order to provide a service.

The above two scenarios have been identified because each of them nicely fits parts of the IIS architecture.

5.3.4.1 A component providing and wholly containing a service

This first scenario deserves recognition as a distinct case because, in many instances, it provides a powerful way of conceptualizing systems, e.g. in layered communication systems such as OSI, and indeed the stratified communication support system (C) in the IIS itself.

Looking back to figure 5.1 we can see that we have portrayed the IIS Communications Complex as three interacting objects: PS, SE and CM. Our diagrammatic representation in figure 5.1 of the composite structure of the complex is misleading. We focus on our misleading representation of the Communications Complex in figure 5.6. (We ignore the PS service for now, since it is discussed in section 5.3.5 as a special case.)

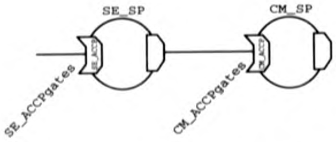


Figure 5.6: An incorrect representation of the SE/CM structure

The FRB implies that the *service definition*[8] of the SE_Service is provided by the specification at the SE_ACCP interface of the SE_Service_Provider.

Figure 5.6 does not unambiguously depict this fact; a reader of figure 5.6 may be led to believe that the specification of the SE_ACCP interface of the SE_SP cannot alone provide a *complete* service definition of the SE_Service since what happens at

[8]to use OSI terminology

this interface is dependent on constraints applied at the CM_ACCP interface. In fact, the specification of the SE_ACCP interface already allows for constraints such as those applied by the CM_ACCP interface.

The problem in figures 5.6 and 5.1 lies in the way in which we have chosen to diagrammatically depict the structuring of SE/CM. These diagrams ought to have made it clear that CM is in fact a *sub-component* of SE, and that SE and CM do not exist at the same level of (de)composition. This is clearly shown by figure 5.7. The *SE_SP* is correctly decomposed as a linear chain of separate *components*. The *SE_SP'* represents the residuum *component* when the *CM_SP component* is 'subtracted' from the *SE_SP component*.

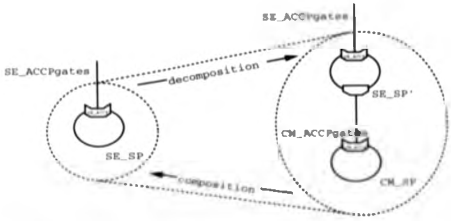


Figure 5.7: A correct representation of SE/CM structure

```
(* Component class: server component *)
(* Comments: an abstraction of SE_Service_Provider server component.
   Illustrates that the CM_SP is a subcomponent of SE_SP, and so that
   SE_Srv_Provider = SE_Srv_Provider_residuum + CM_Srv_Provider. *)
process SE_Srv_Provider [SE_ACCPgates] : noexit :=
   hide CM_ACCPgates in (
         SE_Srv_Provider_residuum[SE_ACCPgates,CM_ACCPgates]
      |[CM_ACCPgates]|
         CM_Srv_Provider[CM_ACCPgates]
   )
endproc (* SE_Srv_Provider *)
```

We can similarly define SD in terms of a composite *component* providing, and wholly containing the service which it offers.

5.3.4.2 A component offering but not wholly containing a service

This second scenario has many instances within the IIS. If we turn our attention to the Business Complex, we find that this *component* is described in terms of the interacting *components* BC, AC and RM. However, these *components* do not interact in a linear-chain fashion to form a stratified *component* as found for SE/CM, but in fact have a cyclic dependency. Indeed, from a wider perspective we see that the whole IIS can be described in terms of interacting (composite) *components* which are inter-dependent.

Let us concentrate solely on the *components* BC, AC and RM for a moment. The BC, AC and RM *components* are described, in FRB, by their *server-rôle interface components* for BC_ACCP, AC_ACCP and RM_ACCP respectively. The FRB also tells us that BC, AC and RM must inter-work with each other and with other IIS *components* in order to fulfill their duties.[9] Hence BC, AC and RM have each a *client-rôle interface component* through which they invoke the services of other *components*. Abstracting from the communications apparatus provided by C, figure 5.8 illustrates the dependency between the three *components*.

From figure 5.8 we can immediately see that an isolated specification of, say, the BC_ACCP *server-rôle interface component*, defines only loosely the behaviour occurring at the BC_ACCP interface. For a more constrained "service definition" of the BC_Service we need to consider the interactions between BC, AC and RM.[10]

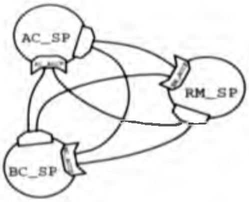


Figure 5.8: Dependency between BC, AC and RM

In this example the direct communication between BC, AC and RM is only an abstraction — a refinement reveals that this interaction is in fact indirect and realized through the PS *component*, the topic of the next section.

[9]This is in contrast to SE which can provide the SE_Service without soliciting help from other *components*.

[10]Note that we are not suggesting that there is anything 'wrong' with the FRB defining BC, AC and RM in this way, but our goal here was merely to clarify this inter-working concept for BC, AC and RM.

5.3.5 The protocol_support_service

We discovered that the level of detail, in the FRB, on the integration of the IIS subsystems was too slight to form a satisfactory reference guide as to how many of these subsystems communicate. The FRB provides only abstract descriptions such as: "entities from the Information, Business and Front-End Service Complexes will communicate with one another through the use of access-protocols (see figure 5.4) which are supported by the underlying Communications Complex". The problem is that X_ACCP communications do not readily map onto SE_ACCP of the the underlying SE_Service. This indicated the need for the existence of some kind of entity which maps X_ACCPs (belonging to B, I and F) to SE_ACCPs. Discussions with CIM-OSA personnel confirmed this hypothesis and the Protocol_Support_Service[11] was born. Figure 5.9 shows an **X_ACCP_PS Service_Provider** in context.

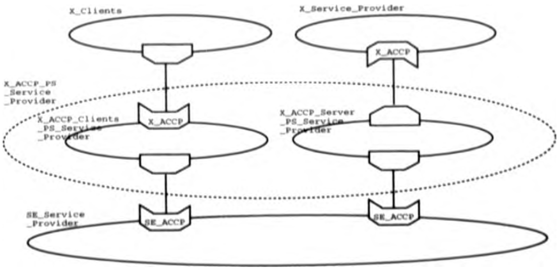


Figure 5.9: An X_ACCP_PS_Service_Provider in context

The following paragraphs provide more detail on the PS_Service.

Providing transparency When an X_Client wishes to use the X_Service, it should only have to initiate the desired **X_Function_Call** and then receive the result of this call. It should not have to deal with any of the underlying communications problems (such as protocol conversion and message transport). Similarly, the X_Service_Provider should only have to receive incoming X_Function_Calls, and return their results, in the

[11] In [McC90a] this was termed the "Stub-Layer", where the use of the term "stub" was proposed because of the similarities of functionality and nature between "stub-entities" in the IIS and the RPC stubs of [BN84].

X_Function_Call format. Therefore some means of supporting transparent communication between X_Clients and the X_Service_Provider must exist. It is this support that is provided by the PS_Service.

Asymmetric nature of the PS_Service The PS_Service is asymmetric in nature. There is the client's side and the server's side (represented in figure 5.9 by the **X_ACCP_Clients_PS_SP** and **X_ACCP_Server_PS_SP**, respectively). The server's side is both different and much more complex in terms of *mapping* functionality than the client's side. Also, we may well define **Particular** X_ACCP_Clients_PS_SPs, such that we define a set of X_ACCP_Clients_PS_SPs, each element of this set offering a different subset of X services to its users. In contrast, there is only one X_ACCP_Server_PS_SP which must support the full range of X services.[12]

Syntactic and semantic mapping It is important to realize that the PS_SPs not only support the syntactic mapping between the X_Layer and the SE_Layer (i.e. mapping X_ACCP PDUs onto SE_ACCP PDUs, and vice versa), but also support a certain amount of *semantic mapping* (i.e. supporting the desired behaviour of the X_Function_Calls through the appropriate choice, use, and ordering of SE_Function_Calls).

The PS_SPs will also handle (transparently to their X_Clients and X_SP) tasks such as: *error management* — actions to be performed if SE reports an error (e.g. retry now, later, not at all, etc.); *concurrency* — an issue for both the X_ACCP_Clients_PS_SP and X_ACCP_Server_PS_SP which may have to schedule a number of concurrent dialogues between X_Clients and the X_SP.

Different X_ACCP specifications Closer examination of the situation portrayed by figure 5.9 reveals that the two interfaces labelled X_ACCP are not quite identical. The FRB labels PDUs handled by the X_ACCP interface of the X_ACCP_Clients_PS_SP as **request** or **confirm** PDUs, and labels PDUs handled by the X_ACCP interface of the X_SP as **indication** or **response** PDUs.[13] Although the FRB *predicts* differences between the contents of request and indication PDUs, and similarly between response and confirm PDUs, it provides no real indication of how or where this difference in contents comes about. This gap in FRB knowledge has been plugged by the introduction of the PS_Service — an introduction brought about by the rigour of formalism.

We imagine the need of a similar Protocol_Support_Service which maps AGEPs from the B, I and F services to suitable SE_ACCPS. Definitions of theses AGEPs have not yet reached a sufficiently stable stage at which to begin formalisation.

5.3.6 Decomposition of an X_ACCP_Clients_PS_SP

The previous subsection provided an overview of the PS_Service, and placed an X_ACCP_PS_SP in context (figure 5.9). In this subsection we describe how we decompose the X_ACCP_Clients_PS_SP *component* of an X_ACCP_PS_SP.

[12]This asymmetry between the client and server sides is also reflected in the RPC paradigm.

[13]Similar labels are found in OSI.

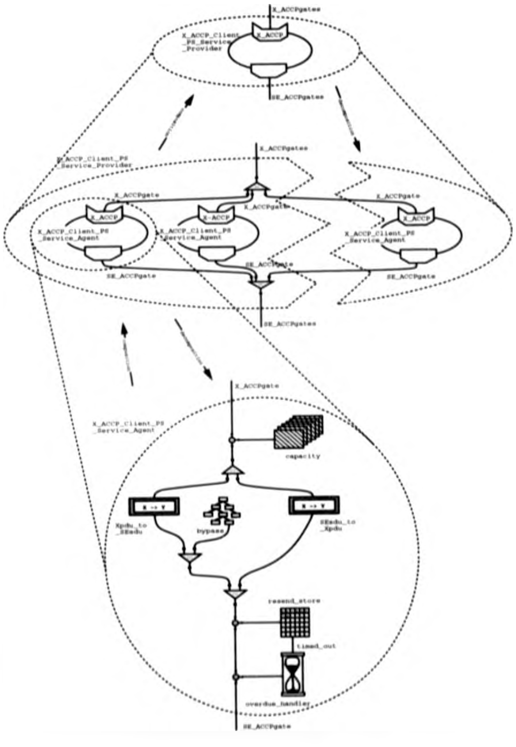


Figure 5.10: Decomposition of an X_ACCP_PS_Client_SP

Firstly, we decompose the X_ACCP_Clients_PS_SP to reveal its distributed X_ACCP_Client_PS_Service_Agents. (This is a more specific example of the decomposition of any X_Service_Provider, which we described in section 5.3.3.) Graphically this decomposition is shown as part of figure 5.10; refer to appendix A.1 for outline XL text for this decomposition.

Secondly, we decompose a single X_ACCP_Client_PS_Service_Agent to reveal it composition in terms of *transformational*, *storage*, *resource-management* and *timeout components* and *combinators*. This structural decomposition in terms of *common architectural components*, depicted by part of figure 5.10, is reflected in XL in appendix A.2.

We believe that this architecturally driven decomposition conveys, in a reasonably understandable manner, the primary *components* of an X_ACCP_Client_PS_Service_Agent.

- The *capacity resource-management component* governs the number of X_ACCP requests that can be dealt with concurrently.
- The *Xpdu_to_SEsdu transformational component* combines X_ACCP PDUs into SE_ACCP SDUs.
- The *SEsdu_to_Xpdu transformational component* breaks SE_ACCP SDUs into X_ACCP PDUs.
- The *bypass functional component* allows the *resend_storage component* to resend a timed out SDU Request, without having to synchronize with the *Xpdu_to_SEsdu component*.
- The *overdue_handler timeout component* notes the time at which each SE_ACCP SDU *Request* is sent. If a corresponding SE_ACCP SDU *Reply* is not received within the *timeout_period* then the *overdue_handler component* will generate a *timed_out* event.
- The *resend_storage storage component* makes a temporary copy of each SE_ACCP SDU *Request* sent. The *resend_storage component* will delete the copy an SE SDU *Request* once an appropriate SE SDU *Reply* has been received within its timeout period. If *resend_storage* receives a *timed_out* event, it resends the appropriate SE_ACCP SDU *Request*.

  The *resend_storage component* may buffer SE_ACCP SDUs for noticeable lengths of time. Notice that the only other *components* to buffer a complete SE_ACCP SDU or X_ACCP PDU are the *Xpdu_to_SEsdu* and *SEsdu_to_Xpdu components*, and these two *components* buffer a single SDU or PDU only long enough to convert package/unpackage it and send it on. Therefore we have localised to the *resend_storage component* all problems of implementing a buffer for storing SDUs for sizable amounts of time.

5.3.7 A revised view of the IIS

This section has provided an insight into how we have decomposed the CIM-OSA IIS, using an architecture driven method. We have used the *common architectural components* defined in chapter 4 to build a skeleton description of the IIS. Building this skeleton description has helped resolve, clarify and better organize aspects of the FRB IIS descriptions.

In the next section, we go part-way towards 'fleshing out' one *component* of the IIS skeleton that we have looked at in this section.

5.3.7.1 Discussion

One important point to come from our consideration of the IIS architecture is that there is no objective way of cutting up the IIS, or viewing any part of it: an X_ACCP can be associated with an X_Service_Provider resulting in a provider-oriented view of the X_ACCP. What looks like a system wide service is actually composed of a complex distributed set of inter-working agents, each of which is responsible for offering the service only to a single system node. An abstraction of a simple client-server communication model hides a much more complex PS based communication model. Elements from abstractions, decompositions and viewpoints may be mixed freely with one another to suit the purpose in hand. The resulting descriptions are 'good' descriptions if they provide a satisfactory description of the system for the purpose in hand.

In our strategy for the formalisation of the IIS, we identified that the second milestone would deliver a suite of formal specifications of IIS elements. In the next subsection, by way of example, we select one such specification, outline its LOTOS description, and list some of the questions which the formalism of LOTOS forced us to recognise. For our example we chose the "service-definition" of the SE — i.e. the SE_ACCP *server-rôle interface* of the SE_Service_Provider in figure 5.7.

5.4 An example IIS specification

The previous section concentrated on constructing an architectural skeleton of the IIS. In this section we select one of the *components* from the skeleton, and examine how to specify it in detail. The specification is structured with the aid of the architectural *components* defined in chapter 4, and employs the special performance features of XL, defined in chapters 6, 7 and 8. Once we have produced our specification, we list some of the questions and answers uncovered by the formalising process.

For our example *component* we chose the the SE_ACCP *server-rôle interface component* (of the SE Service-Provider in figure 5.7).[14] In OSI terminology, this would be called the "service definition" of the SE_Service.

5.4.1 The SE Service

The SE_Service is a complex system. For the purposes of this case-study, we use an abstraction of the SE_Service. This subsection describes those aspects of the SE_Service which are important for our case-study specification. The information in this subsection has been inferred from the current FRB description [CIM89b].

The SE_Service is a conceptual system-wide IIS *component*. Section 5.3.3 shows that a system wide service is really a set of inter working service-agents. However, to provide

[14]Throughout this section we use the term 'SE_Service' to mean 'SE_ACCP *server-rôle interface component*'

a "service definition" abstraction of the SE_Service, we choose to ignore such physical distribution aspects.

The SE_Service must fulfill a number of functional requirements and performance requirements — these are described in the next two subsections.

5.4.1.1 Functional requirements

With respect to functionality, the SE_Service consists of a set of **SE_Callable_Functions** each of which represents a usable communication service provided by SE. Each SE_Callable_Function has a set of *input parameters* and a set of *output parameters*. Input parameters convey the data to be transparently communicated, and the information necessary for the communication (e.g. source and destination addresses, etc.). Output parameters convey the result of a communication (e.g. response data).

Below we describe an example of the use of the SE_Service which (hopefully) will convey the essence of the SE_Callable_Functions, without describing them in detail. Readers are referred to the [CIM89b] FRB items, and to the XL specification in appendix B for a detailed description of SE_Callable_Functions and their associated parameters.

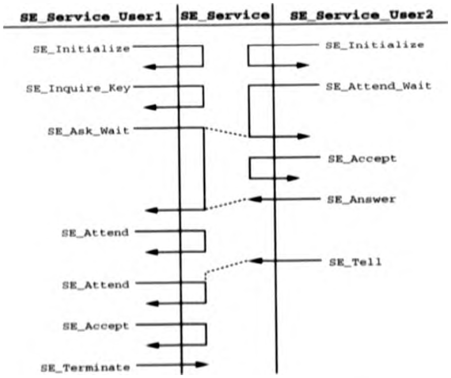


Figure 5.11: Example use of the SE_Service

The event-sequence diagram shown in figure 5.11 illustrates an example use of the SE_Service by two SE_Service_Users. An informal explanation of this event-sequence diagram follows:

1. *User 1* and *User 2* register their interest through the *SE_Initialize* function calls.
2. *User 1* then asks SE to supply it with the *HS_Service_Key* of *User 2* before invoking an *SE_Ask_Wait* function call.
3. Meanwhile *User 2* has invoked an *SE_Attend_Wait* call, which returns (with *output* parameters) when the *SE_Ask_Wait* call arrives from *User 1*.
4. *User 2* then reads the actual transmitted data via the *SE_Accept* call before sending the response to *User 1's SE_Ask_Wait* call via an *SE_Answer* call.
5. When the data from the *SE_Answer* arrives it is conveyed to *User 1* via the *output* parameters of its *SE_Ask_Wait* call invocation.
6. Figure 5.11 then goes on to show a *User 1 SE_Attend* call returning without finding any relevant messages, before a second probe using *SE_Attend* returns to inform *User 1* that it has received an *SE_Tell* message from *User 2*.
7. *User 1* then accepts this message sent via *SE_Accept* before cancelling its interest by issuing an *SE_Terminate*.

The above example is intended to impart an idea of the overall picture of the functioning and purpose of the SE_Service to the reader. The example shows that the SE_Service supports a kind of "passive attention control" [CIM89b] mechanism (and not an "active attention control" mechanism as found in OSI). Under the passive attention control regime an SE_Service_User is not actively informed of the arrival of messages targeted at it, but instead must 'probe' SE (via *SE_Attend_Wait* or *SE_Attend*) for an indication of the arrival of messages.

5.4.1.2 Performance requirements

The FRB is a little vague on performance requirements for the SE_Service. The following list of performance requirements has been inferred from FRB statements — we have classified, and elaborated some of these requirements to make them more understandable.

PR1 Resource management requirement: 'The SE_Service should support a number of multiple concurrent SE_Service_Users and SE_Callable_Function invocations.'

PR2 Quantitative timing requirement: 'The invoker of an *SE_Ask_Wait* function-call can specify the timeout period within which the call must return with some return-status.'

PR3 Probability and quantitative timing requirement: 'The SE_Service guarantees to return a high proportion of all non-waiting SE_Function_Calls (i.e. excluding *SE_Ask_Wait* and*SE_Attend_Wait*), within some specified time limit (the target service-time).'

PR4 Priority requirement: 'Higher priority SE_Function_Calls are accepted before lower priority calls.'

The FRB does not supply actual quantitative data for performance constraints. Nevertheless performance constraints are an aspect of the requirements for the SE_Service and should be reflected in the formal specification of the SE_Service.

5.4.2 Specification of the SE_Service

This subsection develops an XL specification of the SE_Service. Our design process is architecture-driven: we construct the SE_Service using instances of the *common architectural components* defined in sections 4.4.3 to 4.4.8. These *component* instances are customized to fulfill the functional and performance requirements overviewed in sections 5.4.1.1 and 5.4.1.2.

5.4.2.1 The SE_SERVICE component

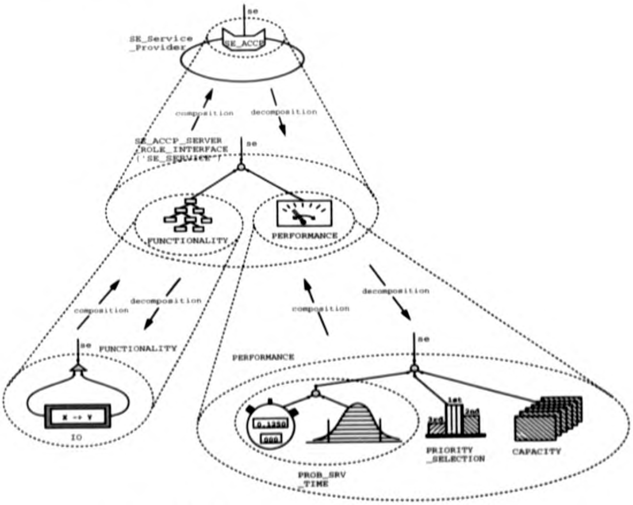


Figure 5.12: Decomposition of the SE_Service (1)

SE_SERVICE is defined as a *server-rôle interface component*. The first decompositional step separates functional requirements from performance requirements, resulting in the *components FUNCTIONALITY* and *PERFORMANCE* (see figure 5.12). This separation of requirements is not perfect. In particular, performance requirements for *SE_Ask_Wait* are quite integrated with the functional requirements, forcing us to specify the performance requirements for this SE_Function within the *FUNCTIONALITY component*.

5.4.2.2 The PERFORMANCE component

The *PERFORMANCE component* is decomposed into three *sub-components*: *PROB_SRV_TIME*, *PRIORITY_SELECTION* and *CAPACITY* (see figure 5.12).

5.4.2.3 The PROB_SRV_TIME component

This *component* captures the quantitative timing and probabilistic requirement PR3 (section 5.4.1.2). To realize this requirement *PROB_SRV_TIME* constrains the *actual service-time* of non-waiting SE_Function_Calls such that *most actual service-times* are within the *target service-time*.

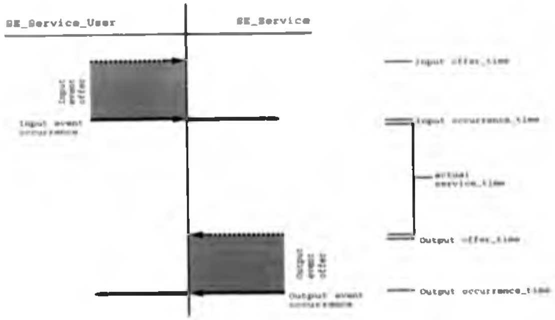


Figure 5.13: A timing breakdown of an SE_Function_Call

To do this *PROB_SRV_TIME* notes, for each non-waiting SE_Function_Call, the *Input* event *occurrence-time*, and then specifies the earliest time at which the corresponding *Output* event is offered (the *Output offer-time*). (When the *Output* event is offered to the SE_Service_User, it indicates that the SE_Service has finished 'servicing' the function call. Then, when the *Output* event occurs, it indicates that the SE_Service_User has accepted and received the function-call — the function call is 'returned'.)

The *actual service-time* equals the *Output offer-time* minus the *Input occurrence-time* (see figure 5.13). *PROB_SRV_TIME* imposes XL quantitative timing and probabilistic constraints to ensure that:

- *actual service-time* $\leq$ *target service-time*, for 99.9% of SE_Function_Calls (corresponding the the "high proportion" mentioned in requirement PR3)
- *actual service-time* $>$ *target service-time*, for 0.1% of SE_Function_Calls.

The *PROB_SRV_TIME component* also enforces the functional requirement of representing each SE_Function_Call by both an *Input* event and an *Output* event. *PROB_SRV_TIME* pairs complementary *Input* and *Output* events by ensuring that each event in a pair contain the same *Caller Key* or *Name*.

5.4.2.4 The PRIORITY_SELECTION component

This *component* captures the requirement for priority PR4. To realize this requirement *PRIORITY_SELECTION* uses XL's priority features to order the occurrence of *Input* events, based upon the priority parameters found in each *Input* SDU.

Note the importance of using XL's priority feature for this task: this XL feature allows *PRIORITY_SELECTION* to preview the priority parameters of a set of *Input* event offers, and order these, before actually accepting the *Input* offers. To do this without the use of XL's priority features would require the construction of an explicit mechanism for sending queries and instructions, concerning priority, to the supplier of *Input* SDUs. Although some such mechanism might be realized at the program coding level, the explicit definition of such a mechanism may be considered as imposing unnecessary constraints at the specification level.

5.4.2.5 The CAPACITY component

This *component* captures the resource management requirement PR1. To realize this requirement *CAPACITY* offers to synchronize on a limited number of concurrent *Input-Output* event pairs. Since each pair represents an SE_Callable_Function invocation, *CAPACITY* limits the total number of concurrent SE_Callable_Functions invocations and, hence, SE_Service_Users.

5.4.2.6 The FUNCTIONALITY component

Abstracting from spatial distribution[15], the SE_Service operates as a *transformational component*. It accepts *Input* events and transforms these to produce *Output* events. Hence we decompose the *FUNCTIONALITY component* into the *transformational component IO* and its *sub-components* (see figure 5.14).

[15]The SE_Service_Provider functions as an *asynchronous communication component*, transmitting messages from one SE_Service_User to another SE_Service_User. However, this case-study ignores the spatial distribution aspects of the SE_Service_Provider to concentrate on its "service definition", i.e. the specification of the SE_ACCP *server-rôle interface component* (the 'SE_Service').

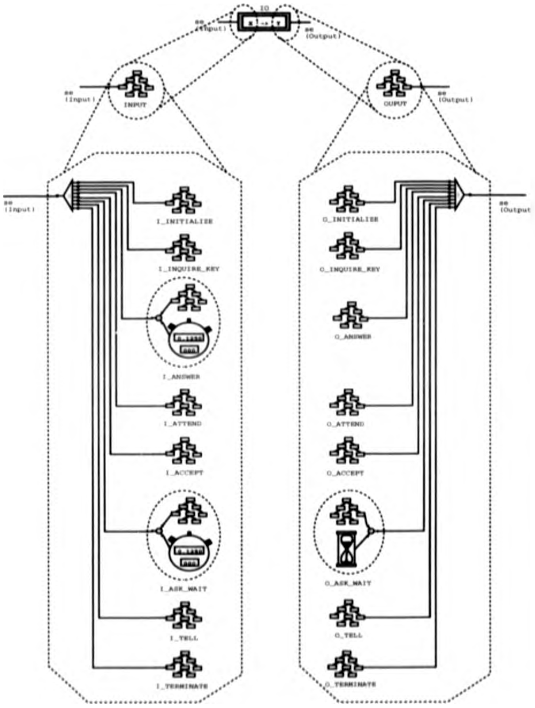


Figure 5.14: Decomposition of the SE_Service (2)

We describe the important aspects of *IO* later, but first we describe how to model an SE_Function_Call.

5.4.2.7 Modelling an SE_Callable_Function

The SE_Service presents the services which it offers to its SE_Service_Users in the form of SE_Function_Calls. All SE_Function_Calls have an *Input* parameter list and an *Output* parameter list. An SE_Function_Call invocation can be modelled by two suitably structured XL events. Two events are used to model the asynchrony between the *Input* and *Output* aspects of a function call.

For example, if we consider the *SE_Ask_Wait* function, the two XL events (with corresponding ACT ONE abstract data types) which model it, will have the following coarse structures:

```
se ! SE_Sdu( (* SE service data unit *)
                SE_Ask_Wait, (* SE callable-function type *)
                Input, (* service primitive *)
                requester_key, (* ID of client *)
                responder_key, (* ID of server *)
                Request_Pdu(data), (* request-data *)
                priority, (* priority of this function call *)
                timeout (* timeout period *)
           )

se ! SE_Sdu( (* SE service data unit *)
                SE_Ask_Wait, (* SE callable-function type *)
                Output, (* service primitive *)
                requester_key, (* ID of client *)
                Response_Pdu(data) (* response-data *)
           )
```

Of course ACT ONE data types with appropriate constructor, selector, etc. operations and data values must also be specified to support the above model.

5.4.2.8 The IO component

IO functions as a *transformational component* which forms appropriate *Output* events given a previous history of *Input* events.

IO has two *sub-components*:

- *INPUT* which accepts *Input* events and updates *history information* accordingly
- *OUTPUT* which offers suitable *Output* events, given the current *history information*.

History information consists of a set of data structures which capture all important aspects of the previous history of SE_Callable_Function invocations. The purposes of these data structures are described below.

- *reqset* (read as 'request message set') contains messages which have been given to the SE_Service via *SE_Tell* or *SE_Ask_Wait* function calls, and which have yet to be delivered[16] to their target SE_Service_Users.

- *rmset* (read as 'response message set') contains similar 'yet to be delivered' messages which have been submitted to the SE_Service via ***SE_Answer*** calls in response to ***SE_Ask_Wait*** calls.
- *rgset* (read as 'registration set') contains HS_Service_Key/HS_Name pair entries which effectively register an HS entity as an SE_Service_User.
- *outstaskwaits* (read as 'outstanding ***SE_Ask_Wait Output*** offers') indicates the ***SE_Ask_Wait*** invocations which have ***Output*** event offers still to be generated, and contains the information from ***SE_Ask_Wait Inputs*** which will be used to generate these ***Output*** event offers.
- *outstinquires*, *outstattends* and *outstaccepts* have purposes similar to *outstaskwaits*.

The event-sequence graph in figure 5.15 helps explain the relationships between the *history information* data structures and SE_Function_Call events.

5.4.2.9 The O_ASK_WAIT component

The functional and timeout (performance) requirements for ***SE_Ask_Wait*** function calls are quite integrated, and this *component* is responsible for specifying both. (We managed, for the other SE_Function_Calls types, to separate functional from performance requirements at an earlier level of decomposition.)

The *O_ASK_WAIT component* captures the quantitative timing requirement PR2 (section 5.1.1.2). Under this requirement there are three different cases in which an ***SE_Ask_Wait Output*** event may occur. Given an ***SE_Ask_Wait Input***:

1. An appropriate response packet to the ***SE_Ask_Wait Input***, arrives before the ***SE_Ask_Wait*** invocation timeout period expires. In this case the ***Output*** event, with return-code ***SE_Ok***, is offered from the arrival time of the response packet (see figure 5.15).
2. The timeout period expires before an appropriate response packet arrives, but then an appropriate response packet arrives before the ***SE_Ask_Wait Output*** event occurs. The ***Output*** event, with return-code ***SE_Timeout***, is offered from the end of the timeout period.
3. The timeout period expires before an appropriate response packet arrives, and no appropriate response packet arrives before the ***SE_Ask_Wait Output*** event occurs. The ***Output*** event, with return-code ***SE_Timeout***, is offered from the end of the timeout period.

Note that case 2 and case 3 are considered different for the purposes of specification — they cannot be separately identified by examining the ***SE_Ask_Wait Output*** events that

[16] i.e. transmitted to the targeted SE_Service_User's node and read by this SE_Service_User through the use of an ***SE_Accept*** function call

they offer. In XL specification terms, case 2 and case 3 are different because their XL selection-clauses need to be radically different (see the *O_ASK_WAIT* process definition in appendix B).

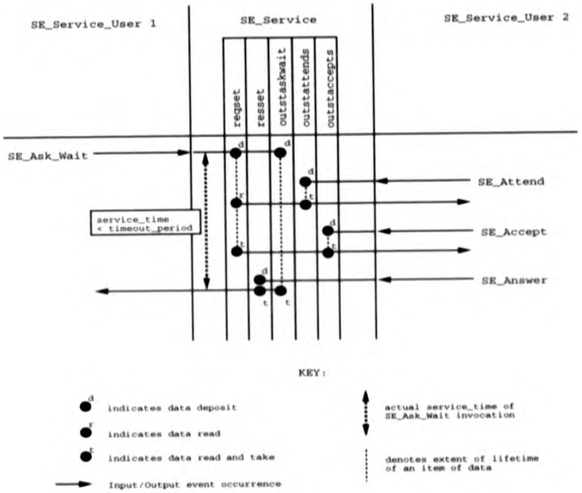

SE_Service_User 1
SE_Service
SE_Service_User 2
reqset
resset
outstaskwait
outstattends
outstaccepts
SE_Ask_Wait
SE_Attend
service_time < timeout_period
SE_Accept
SE_Answer
KEY:
indicates data deposit
indicates data read
indicates data read and take
Input/Output event occurrence
actual service_time of SE_Ask_Wait invocation
denotes extent of lifetime of an item of data

Figure 5.15: How history data structures are related to function calls

To calculate if a timeout should occur during an *SE_Ask_Wait* function invocation, we require three time values:

- *invoke_time* which is the occurrence time of the *SE_Ask_Wait Input* event. This is captured within the *I_ASK_WAIT component*.
- *timeout_period* for the particular *SE_Ask_Wait* invocation. This is specified by the SE_Service_User as a parameter within the *Input* event.
- *res_arr_time* which is the arrival time of an appropriate response packet. This is captured within the *I_ANSWER component*. For the purposes of this case-study, we assume that the arrival time of a response packet corresponds to the time at

which the response packet becomes available to the *SE_Ask_Wait* invocation, at which it is targeted.

5.4.2.10 Example questions arising from formalising the FRB description of SE_Service

As supporting evidence of the benefits of formalism for CIM-OSA, we present a selection of questions and issues which arose from the first attempts at developing an (X)L specification[17] for the SE_Service.

The following list of problems is incomplete and will, we expect, be added to, before eventually being totally resolved through further development work on SE_Service.

Q1. The FRB description says very little about what *definable* behaviour is — this is a very important issue since it will basically decide the extent of any formal specifications for the SE_Service. For this first specification attempt, we took the view (after consulting CIM-OSA members) that *definable* behaviour should include both *valid* behaviour and *expected erroneous* behaviour (e.g. behaviour on message timeout).

Q2. The FRB mentions little about error handling. In particular we are thinking about what constitutes an error (see the above point), what action should be taken when an error occurs, and what the content and format of error reports should be for SE_Function_Calls. Also, should error handling be deferred to a lower-level specification?

OSI models only *valid* behaviour on the basis that other behaviour is erroneous and needs to be handled in an implementation dependent way. Specifications which model, to some degree, *erroneous* behaviour are often considered more concrete. An advantage of specifying *expected erroneous* behaviour error handling is that these would tend to lead to more robust systems. Also, we could argue that certain 'error responses' should be at the same level of abstraction as 'normal data responses'.

The specification in this section models some *expected erroneous* behaviour. The present FRB description mentions very little about what *expected erroneous* behaviour is and how the SE_Service handles it.[18]

This decision is reflected in the specification by allowing any *Input* event of the correct structure to occur, thus modelling the SE_Service's acceptance of both *valid* and *expected erroneous Input* events. The XL specification then describes how, if the SE_Service finds an *Input* event representing some kind of erroneous SE_Function_Call invocation, it will decide upon appropriate action.

[17] This specification was mainly derived from the FRB documents: [CIM89b].

[18] Maybe the FRB is deliberately vague on this matter to hint that the handling of erroneous behaviour is entirely an implementation-dependent issue. However CIM-OSA personnel thought that "some" *expected erroneous* behaviour should be modelled. Thus, we have had to improvise the specification of *expected erroneous* behaviour, based on discussions with CIM-OSA personnel.

Q3. Does the SE_Service check the 'validity' of HS_Service_Key arguments in callable functions, and if so what should be the format of the returned error report? (Note that if the HS_Service_Key of the Calling entity is invalid the SE_Service cannot return an error report since it has no means of discovering the correct identify of the Calling entity.) More generally, is it the responsibility of the SE_Service to check other SE_Function_Call argument types?

Q4. In the present FRB descriptions of the SE_Service, only some SE_Function_Call types are confirmed in the sense that the SE_Service_User is informed of the return status of the SE_Function_Call invocation (e.g. the *SE_Ask_Wait* will return with either response data or with a "timeout occurred" indication). However, we perceived the need for some kind of confirmation for all SE_Function_Call types.

Adopted proposal: We decided that all SE_Function_Call types return a 'return code' (e.g. *SE_Ok, SE_InvalidKey, SE_Timeout*) which indicates to the Calling SE_Service_User the status of the SE_Function_Call invocation. This means that all SE_Function_Call types now have an *Output* parameter list which contains at least this return-code parameter. This proposal is implemented in the specification in appendix B.

Confirmed SE_Function_Calls allow *expected erroneous* behaviour (see Q2) to be reported to the function invokers. Also, now that each SE_Function_Call is delimited by an *Input* event and an *output* event, we can attach quantitative timing and probabilistic constraints to these events to meet requirements for SE_Function_Call service time, priority ordering, etc. (see section 5.4.1).

Q5. The FRB states that *SE_Accept* and *SE_Answer* functions should contain *Transaction_Id* parameters which allows SE_Service_Users to identify the function calls which compose each transaction. Our question is, can a single SE_Service_User engage in more than one *SE_Ask_Wait* at any time? Surely not, considering the suspending nature of the *SE_Ask_Wait* function.[19] If an SE_Service_User cannot engage in more than one *SE_Ask_Wait* at any given time then surely the need for a Transaction_Id parameter in *SE_Accept* and *SE_Answer* is redundant: the HS_Service_Key of the *SE_Ask_Wait* initiator (the Calling entity) is all that is required to facilitate the Called entity to respond, in a transaction based nature, to the Calling entity. The *SE_Ask_Wait* initiator can be engaged in only one *SE_Ask_Wait* transaction at any given time, therefore any *SE_Answer* reply can be unambiguously identified by the HS_Service_Key of the *SE_Ask_Wait* initiator.

If an SE_Service_User wishes to distinguish between its SE supported transactions then surely it is the responsibility of that SE_Service_User to somehow locally label its transactions uniquely.

Assumption: The specification in appendix B assumes that an SE_Service_User cannot engage in more than one *SE_Ask_Wait* at any given time, therefore making the Transaction_Id redundant — it is therefore not modelled in this specification.

[19]In figure 5.15 the SE_Service_User 1 is 'suspended' after the *Input* awaiting the occurrence of the *Output* event.

Q6. From the FRB informal description it was unclear on what basis the SE_Service delivered messages. This issue should be addressed in three areas:

- Is receipt ordering of 'messages' in transit the same as submission ordering?
- Under what regime should messages (of the same *Type*) be delivered from the 'arrived message buffer' to the Called entity? Should we use a "first in, first out" (FIFO) scheme, for example?

  In light of these two uncertainties, the XL specification makes a non-deterministic choice when dealing with message delivery.
- When a message is 'delivered' to the targeted SE_Service_Agent node, is it immediately available for delivery to the targeted SE_Service_User?

  The XL specification assumes that this is the case — see the *I_Answer* process in appendix B.

On a similar note, we found that the FRB says very little about message loss, corruption, duplication, misdelivery, etc.

Q7. The FRB definition of an HS_Service_Name is that it is an identifier which is unique within the local node of the Caller. In our specification the HS_Service_Name is represented by *LocalNameType*, but since the SE_Service specification describes a system-wide service, an SE_Service_User must be uniquely identifiable at the SE_Service interface by its "name" (when the SE_Service_User does not yet possess an HS_Service_Key). The *GNameType* realizes this uniqueness property of the "name" by pairing a *LocalNameType* value with an *NodeNameType* value to form a system-wide unique identifier.

5.4.3 Discussion

Structuring the SE_Service specification in terms of chapter 4's architectural *components* has a number of benefits. The graphical representation (figures 5.12 and 5.14) shows the major functional and performance components at-a-glance. The classification of *components* explicitly separates concerns and so aids understanding. The classification of *components* proves useful for navigating around a large design, and finding *components* of concern. For example, this is especially useful for maintenance or experimentation work where the 'bits that need tweaking' can be readily identified.

An XL specification organized in architectural terms is easier to read and to understand. Building specifications in terms of architectural *components* removes some of the (difficult) creative aspects of the specification task. This is because architectural *components* embody some domain knowledge (know how from a previous history of solutions); this is an example of 'knowledge re-use'.

Notice that the SE_Service specification (described in this section) provides an example of how to write a constraint-oriented [VSvSB90] specification (organized in terms of architectural *components*), while the outline X_ACCP_Clients_PS_SP specification (described in section 5.3.6, figure 5.10) provides an example of how to write a resource-oriented [VSvSB90] specification (organized in terms of architectural *components*).

5.5 Guidelines for further development

The final task within our strategy for formalising the IIS is to produce guidelines for the development and assessment of XL IIS descriptions. In part, these guidelines take the form of a specification *development map*. A development map consists of a set of nodes which are (references to) XL specifications, and a set of arcs which are (formal) relations between the specifications. A specification development map helps to explain the relationships between (the XL descriptions of) the facets of the IIS. It provides a guide to the suite of IIS XL specifications (such as those of in the previous sections), suggesting possible development paths. It also provides a framework for conformance testing.

We use this section to take a brief look at a specification development map for a particular subsystem of the IIS.

5.5.1 An example: X_Service development map

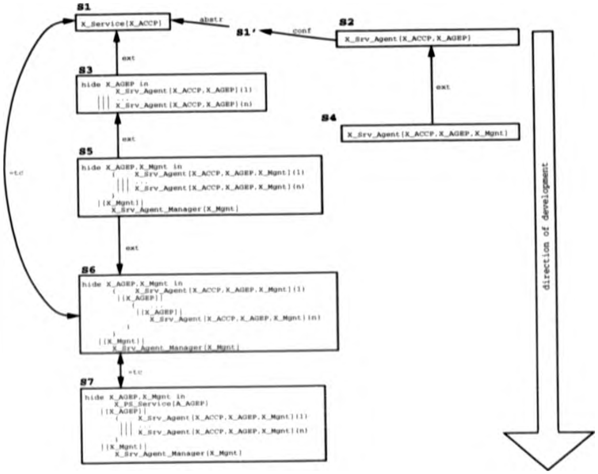
As an example we concentrate on the development of the decomposition of an *X_Service* as portrayed in figures 5.5 and 5.9. Figure 5.16 depicts a simplified, but typical specification development map for the decomposition of the *X_Service*. The following paragraphs provide a short commentary about figure 5.16.

S_1: In the typical development of an *X_Service*, we begin by writing a constraint-oriented style "service definition" (denoted by specification S_1).

S_1 and S_2: Then we proceed towards the goal of a decomposition of the *X_Service* in terms of *X_Service_Agents* (see section 5.3.3) by developing the specification of an *X_Service_Agent* (specification S_2). To build confidence in our S_2 specification, we check that S_2 **conf** S_1', where S_1' is an abstraction of S_1 formed from only those parts of S_1 whose functionality is (supposed to be) reflected in S_2.

S_3: An S_3 specification is a composition of S_2 specifications. This composition takes us towards an emulation of an *X_Service* formed by a set of *X_Service_Agents*. However, these *X_Service_Agents* do not communicate with one another — their *X_AGEP* interfaces remain unconnected. These unconnected interfaces give rise to unwanted behaviour along the X_ACCP system-wide interface (e.g. unwarranted response SDUs, etc.). For this reason S_3 **ext** S_1 (approximately, this means that: S_3 *contains* and extends the functionality of S_1).

S_4 and S_5: The S_4 specification extends the S_2 *X_Service_Agent* specification by adding some *X_Service_Agent* management functionality. S_4 specifications are then composed together with an *X_Service_Agent_Manager* to form the specification S_5. S_5 extends S_4 by the addition of the management functionality.



```
X_Service[X_ACCP]
```

```
X_Srv_Agent[X_ACCP,X_AGEP]
```

```
hide X_AGEP in
         X_Srv_Agent[X_ACCP,X_AGEP](1)
    |||  ...
         X_Srv_Agent[X_ACCP,X_AGEP](n)
```

```
X_Srv_Agent[X_ACCP,X_AGEP,X_Mgnt]
```

```
hide X_AGEP,X_Mgnt in
        (     X_Srv_Agent[X_ACCP,X_AGEP,X_Mgnt](1)
         |||  ...
              X_Srv_Agent[X_ACCP,X_AGEP,X_Mgnt](n)
        )
    |[X_Mgnt]|
        X_Srv_Agent_Manager[X_Mgnt]
```

```
hide X_AGEP,X_Mgnt in
        (     X_Srv_Agent[X_ACCP,X_AGEP,X_Mgnt](1)
          |[X_AGEP]|
              (  ...
               |[X_AGEP]|
                  X_Srv_Agent[X_ACCP,X_AGEP,X_Mgnt](n)
              )
        )
    |[X_Mgnt]|
        X_Srv_Agent_Manager[X_Mgnt]
```

```
hide X_AGEP,X_Mgnt in
        X_PS_Service[A_AGEP]
    |[X_AGEP]|
        (     X_Srv_Agent[X_ACCP,X_AGEP,X_Mgnt](1)
         |||  ...
              X_Srv_Agent[X_ACCP,X_AGEP,X_Mgnt](n)
        )
    |[X_Mgnt]|
        X_Srv_Agent_Manager[X_Mgnt]
```

Figure 5.16: A specification development map for an *X_Service*

S_6: In specification S_6 we compose S_4 *X_Service_Agents* in the fashion illustrated in figure 5.5. The *X_Service_Agent X_AGEP* interfaces are fully interconnected. This restricts the unwanted behaviour we described for S_3 (we no longer have the unconnected interfaces found in S_3), and so S_6 ought to be testing congruent to the *X_Service* specification S_1.

S_7: Specification S_7 maps the 'logical' connections found between the *X_AGEP* interfaces of the *X_Service_Agents* of specification S_6, to connections to the *X_Protocol_Support*. The basis for this decomposition is described in section 5.3.5.

Figure 5.16 is a simple example of a *specification development map*. Such a map can be used to steer the development of the specification, act as a guide to a suite of specifications, and provide a framework for conformance testing.

5.6 Summary

This chapter presented the CIM-OSA IIS as a case-study of architecture-driven specification using XL.

The CIM-OSA IIS is a large, fairly typical distributed computing system which includes both functional and performance requirements. These considerations made the IIS a suitable candidate for testing the worth of chapter 4's architecture framework for distributed systems, and for testing the descriptive power of XL's performance features developed in chapters 6, 7 and 8.

Section 5.3 showed how chapter 4's *common architecture components* could be used to build a skeleton architecture for the IIS. Then section 5.4 focussed on one part of this skeleton to show how the *common architecture components* could be customized, and performance features of XL used, to specify the SE_Service. XL allowed quantitative timing, probabilistic and priority requirements to be expressed and composed easily.

The classification of architectural components in chapter 4 provided guidelines for structuring the (de)composition of the IIS. The resulting XL specifications were organized in terms of architectural components, rather than solely in terms of specification language concepts. The direct reflection of problem domain structure in the specification made it it easier to understand and navigate through the formal specification.

CIM-OSA itself is still in its infancy as regards formalism, but many benefits have already been reaped. In the main, we found that the *rigour* of formalism promoted early problem identification. In the short time that LOTOS[30] has been employed in the CIM-OSA project, it has had a considerable impact, and has gained project-wide acceptance as an integral part of the development of the IIS reference architecture. Our initial use of LOTOS has shown that the so called "Formal Reference Base" IIS descriptions are neither as rigorous nor as complete as required. The application of LOTOS has led not only to design corrections, but also to the development of better designs. Development of LOTOS descriptions has helped to make coherent the description of the IIS. Before the use of LOTOS the descriptions of individual IIS services were not entirely compatible, and this was not immediately evident from examining the informal FRB descriptions.

In conclusion, the CIM-OSA case-study in this chapter has provided a testing ground for the concepts and language extensions defined in chapters 4, 6, 7 and 8.

[30] Note, the CIM-OSA project actually used LOTOS to formalise the IIS, but to demonstrate XL's ability to express performance requirements, we, in this thesis, elaborated CIM-OSA LOTOS specifications to XL specifications — see section 5.1.3.

Chapter 6

Formal specification of timing for distributed systems

This chapter is concerned with the language support required for the formal specification of quantitative timing concerns in distributed systems. We begin by examining the inadequacies of standard LOTOS in this area (using examples from CIM-OSA). We investigate requirements for the expression of quantitative timing concerns, and distill a set of features which we believe a supporting language should include. Then a *time-extended* derivative of LOTOS (TLOTOS) is proposed which unifies and incorporates many of the afore mentioned features. In this time-extended version of LOTOS, the notion of quantitative, *physical-clock* based time is implicit and time constraints are easily expressed. We detail the syntactic and semantic extensions which take us from LOTOS to TLOTOS. Also, we explore two functions for mapping TLOTOS to LOTOS. Neither syntactic function is completely satisfactory — giving weight to the need for semantic-level time extensions to LOTOS, such as developed in this chapter. We conclude by returning to the CIM OSA examples to demonstrate the power of TLOTOS for the capture of quantitative timing requirements. Appendix G forms an annex to the chapter, to show how TLOTOS specifications can be tested under extended definitions of the LOTOS testing relations, to yield sensible and intuitive results.

6.1 Introduction

We often find that real-life distributed systems display time-dependent behaviour. In order to fully specify such time-dependent systems we must use a description language which fully supports this aspect of their behaviour, i.e. the expression of quantitative timing concerns. [CPW86] caution that to omit quantitative timing requirements may subtract an entire dimension from the description. Such omission may prove a useful abstraction for certain tasks, but often the time-dependent aspects of a system are where the real complexity of distributed systems can be found. We must ensure that our description language effectively reflects the time-dependent essence of many distributed systems problems, and does so in a way which is convenient and understandable to the user.

Absence of appropriate timing information can result in ambiguous or erroneous descriptions; but the cumbersome expression of timing information will produce descriptions which are difficult to understand.

Concern that timing information is somehow implementation detail, and should not be a specification level concern, is addressed by [CPW86] which says that: "the apparent distinction between the measures of time introduced for the mathematical concern of *correctness* and those introduced for the engineering concern of *performance* may be wholly illusory because what we originally perceived as a performance concern impacts on correctness".

Process algebras have proved useful in capturing descriptions of complex, concurrent, communicating systems. LOTOS is one such algebra. The formal basis of LOTOS provides it with the combined descriptive and analytic power necessary to tackle such complex systems. However LOTOS lacks the built in facility to express quantitative time, which explains our efforts to form a time-extended derivative of LOTOS for the description and analysis of time-dependent systems.

The next section substantiates, by means of examples, our criticism of informal or expressively cumbersome quantitative time-models.

6.2 The inadequacy of standard LOTOS for expressing timing concerns

This section examines some real-time aspects of the CIM-OSA IIS, and explains why standard LOTOS is inadequate for their description. We use the CIM-OSA IIS X_Service and X_Service_Agent specifications as our example subjects (see section 5.5), and examine abstractions of these which emphasize timing aspects.

This section has the following structure: we begin with informal descriptions of the X_Service and X_Service_Agent. Then we provide LOTOS descriptions of these systems, one using an informal model of time and the other using a formal but cumbersome model of time. We discuss the inadequacies of these LOTOS time-models, and propose the development and use of a time-extended version of LOTOS.

6.2.1 Informal descriptions of CIM-OSA IIS timing aspects

The following two subsubsections provide informal descriptions of the timing essentials of the X_Service and X_Service_Agent.

6.2.1.1 Informal description of the X_Service

The following description captures one aspect of the timing essence of an X_Service. (The description is really an abstraction of an X_Service, which suffices for the purposes of this section.)

- The X_Service is willing to accept a *Request* at any time (say *t1*), at the X_ACCP system wide interface. The X_Service does not place any quantitative timing constraints on this event.
- A *Request* event results in the X_Service offering a complementary *Response* event at the X_ACCP interface. This event will be attributed with either *data2* (the result of some computation), or the value *Timeout* (indicating that a timeout has occurred).
- If a *data2 Response* is offered, its offering will begin in the time range *t1* ... (*t1* + *timeout_period*), inclusive. A further requirement is that the X_Service is to compute and then offer a *data2 Response* as quickly as it can, i.e. the X_Service is to offer this event ASAP (as soon as possible).

  Otherwise, if the *Timeout Response* is offered, its offering will begin at the time *t1* + *timeout_period* + 1. In other words, if the X_Service cannot offer the *data2 Response* within the *timeout_period* then a *Timeout Response* will be offered immediately after the end of the *timeout_period*.

  The X_Service is responsible for deciding which *Response* event is offered, and at what time it is offered. Also, the X_Service is responsible for deciding the value of *data2*.

6.2.1.2 Informal description of the X_Service_Agent

The following description captures the timing essence of an X_Service_Agent, and should be read in conjunction with the description of an X_Service in section 6.2.1.1. (Again, this description is an abstraction of an X_Service_Agent, constructed for the purposes of this chapter.)

- To perform a *Request*, the X_Service_Agent solicits the help of other X_Service_Agents via the *X_AGEP* interface (see section 5.3.3). An X_Service_Agent should be willing to participate ASAP (as soon as possible) in events at the *X_AGEP* interface. This requirement reflects the urgency indicated for *data2 Response* events in the X_Service definition above.

6.2.1.3 Discussion

Our concern in this section lies not in illuminating the general disadvantages of informal description but with the inadequacies of informal or cumbersome time-models, which may themselves exist within formal descriptions. However the above requirement statements typify the way in which real-time system requirements may be ill conceived or vaguely stated in an informal language. Of course, "requirements-level" descriptions are necessarily vague (or abstract) — we cannot embody all the knowledge we know about the universe of discourse of the problem into a brief requirements-level description. In general, no aspect of the requirements enjoys unambiguous interpretation, but timing requirements, being of a precise nature, often tend to suffer adversely in "requirements-level" *abstractions*.[1]

We assume an unambiguous interpretation of these requirement statements in the production of the formal descriptions of the following subsections.

6.2.2 An informal time-model

By an informal time-model, we mean that no definition exists for a formal framework within which to reason about quantitative time. This makes it impossible to formally reason about relationships between statements concerning quantitative time. This leaves the quantitative timing aspects of the description open to subjective interpretation. This we demonstrate through the following examples.

6.2.2.1 The X_Service using an informal time-model

The following LOTOS description of the X_Service is based on an informal time-model.

```
(* Specification of a limited X_Service, in LOTOS, focussing on timing aspects *)

process X_Service[X_ACCP] : noexit :=
    X_ACCP ! Req ? data1:DataSort (* occurs at any time t1 *);
    (
        choice data2:DataSort []
            ([data2 ne Timeout]->
                i (* should occur at a time t2 <= t1+timeout_period *)
                  (* and ASAP within this time period *);
                X_ACCP ! Res ! data2 (* should occur at a time >= t2 *);
                stop
            )
    []
        i (* should occur at a time t3 = t1+timeout_period+1 *) (* timeout *);
        X_ACCP ! Res ! TimeOut (* should occur at a time >= t3 *);
        stop
    )
endproc (* X_Service *)
```

[1] The general rule is that a ("requirements-level") description should be as complete as necessary for some particular purpose.

- Quantitative timing constraints are expressed as comments in the LOTOS text. (These constraints have no formal basis.)
- The **i** events indicate that the X_Service is responsible for deciding which *Response* event is offered, and (in principle) its occurrence time. The **i** event within the *sum-expression* is also used to indicate that the X_Service is responsible for deciding the value of *data2*.

6.2.2.2 The X_Service_Agent using an informal time-model

The following LOTOS description of the X_Service_Agent is based on an informal time-model.

```
(* Specification of a limited X_Service_Agent, in LOTOS, focussing on timing aspects. *)

process X_Service_Agent[X_ACCP,X_AGEP]  noexit :=
    X_ACCP ! Req ? data1:DataSort (* occurs at any time t1 *);
    (
        X_AGEP ! Req ! data1 (* should occur at a time <= t1+timeout_period *)
                             (* and ASAP within this time period *);
        X_AGEP ! Res ? data2:DataSort [data2 ne Timeout]
                             (* should occur at a time t2 <= t1+timeout_period *)
                             (* and ASAP within this time period *);
        exit(data2) (* should occur at time t2 *)
    [>
        i (* should occur at a time t3 = t1+timeout_period+1 *) (* timeout *);
        exit(TimeOut) (* should occur at time t3 *) (* TimeOut is in DataSort *)
    )
    >> accept data2:DataSort in
        X_ACCP ! Res ! data2 (* occurs at any time *);
        stop
endproc (* X_Service_Agent *)
```

- We ensure that **exit** events consume no time, by commenting that each of these should occur at the same time as its immediately preceding event.
- The X_Service_Agent soliciting help from X_Service_Agents is represented by the events at the X_AGEP interface.
- A timeout may occur at any state in the event sequence:

$$\rightarrow X\_AGEP!Req \rightarrow X\_AGEP!Res \rightarrow \textbf{exit}$$

  This is realized by allowing the **i** *(* timeout *)* event to disable ([>) this sequence. However, from the above description, it is not necessarily obvious that the **i** *(* timeout *)* event may occur after the *X_AGEP!Res* event. One interpreter might argue that the choice between the **exit***(data2)* and **i** *(* timeout *)* events is deterministic in that the **exit***(data2)* pre-empts the occurrence of the **i** *(* timeout *)* event, because the **exit***(data2)* can only occur at an earlier time than the **i** *(* timeout *)* event.

- With respect to timing, we have no basis for formally proving that the X_Service_Agent description is equivalent to the X_Service description.
- We have restricted the *data2* parameter in the *X_AGEP!Res* event, so that it cannot include the term *Timeout* of the sort *DataSort*. In a real IIS system, we would expect that this parameter could contain the value *Timeout*, indicating that the object whose help has been solicited via the X_AGEP!Req call has itself timed-out and returned this fact via the X_AGEP!Res parameter. However for our example, we disable the possibility of the solicited object returning a *Timeout* parameter, to ensure that a solicited object timeout cannot be mistaken for a timeout of the X_Service_Agent in question (given that it is the difference between the *Timeout* behaviour and any alternative behaviour of the X_Service_Agent that we are especially interested in).

6.2.2.3 Discussion

The above informal time-model based descriptions of the X_Service and X_Service_Agent are inadequate. Without formal semantics there is no means of enforcing precise behaviour, nor can we "prove" any of the quantitative timing aspects. As writers of the specification, we know what we mean by the comments concerning quantitative time, but this does not guarantee an objective interpretation by everyone.

These examples illustrate a few of the problems which can arise if we have no formal framework in which to reason about statements about quantitative time.

6.2.3 A formal, but inadequate and cumbersome time-model

For a time-model, more formal than the model in the previous subsection, we choose '*t*-events' to represent the passing of units of time. Now consider the description of the X_Service_Agent using this time-model.

6.2.3.1 The X_Service_Agent using an more formal time-model

The following LOTOS description of the X_Service_Agent is based on the *t*-event time-model.

```
(* Specification of a limited X_Service_Agent, in LOTOS, using explicit t-events *)

process X_Service_Agent[t,X_ACCP,X_AGEP] noexit :=
    State1[t,X_ACCP,X_AGEP]
where
    process State1[t,X_ACCP,X_AGEP] noexit :=
            t; State1[t,X_ACCP,X_AGEP]
        [] X_ACCP ! Req ? data1:DataSort; State2[t,X_ACCP,X_AGEP](0, data1)
    endproc (* State1 *)

    process State2[t,X_ACCP,X_AGEP](timer:Nat, data1:DataSort) noexit :=
            ([timer le timeout_period] -> t; State2[t,X_ACCP,A_AGEP](Succ(timer)))
        [] ([timer le timeout_period] -> X_AGEP ! Req ! data1 (* and ASAP *);
                                        State3[t,X_ACCP,X_AGEP](timer))
    [] ([timer eq timeout_period+1] -> i (* timeout *); State6[t,X_ACCP])
  endproc (* State2 *)

  process State3[t,X_ACCP,X_AGEP](timer:Nat) noexit :=
        ([timer le timeout_period] -> t; State3[t,X_ACCP,A_AGEP](Succ(timer)))
    [] ([timer le timeout_period] -> X_AGEP ! Res ? data2:DataSort
                                          [data2 ne Timeout] (* and ASAP *);
                                        State4[t,X_ACCP,X_AGEP](timer, data2))
    [] ([timer eq timeout_period+1] -> i (* timeout *); State6[t,X_ACCP])
  endproc (* State3 *)

  process State4[t,X_ACCP](timer:Nat, data2:DataSort) noexit :=
        (i; State5[t,X_ACCP](data2))
    [] ([timer eq timeout_period+1] -> i (* timeout *); State6[t,X_ACCP])
  endproc (* State4 *)

  process State5[t,X_ACCP](data2:DataSort) noexit :=
        t; State5[t,X_ACCP](data2)
    [] X_ACCP ! Res ! data2; stop
  endproc (* State5 *)

  process State6[t,X_ACCP] noexit :=
        t; State6[t,X_ACCP]
    [] X_ACCP ! Res ! Timeout; stop
  endproc (* State6 *)

endproc (* X_Service_Agent *)
```

It is not easy to understand the above LOTOS description at a glance. The FSM in figure 6.1 may help clarify the LOTOS description (the *timeout_period* is given the value 3 in this FSM representation).

- The requirements said that an X_Service_Agent was to begin offering a *Request data2* event at a time within the *timeout_period*, or begin offering the *Request Timeout* event at the time immediately after this period. Looking at the FSM, it is obvious that this is the case. The processes *State5* and *State6* of the LOTOS description, correspond to the states described by this requirement. These two states are reached at the correct times (count the *t*-events), and the appropriate *Response* events are offered at these two states.

- The **i** event, in processes *State4*, directly preceding the *process-instantiation*, *State5...*, corresponds to the **exit** *(data2)* event from the X_Service_Agent description in the previous subsection. The occurrence of this event represents the point in the behaviour of the X_Service_Agent at which it is ready to offer the *Response data2* event. This **i** event has no *t*-event alternative because, according to the X_Service_Agent description in the previous subsection, the **exit** *(data2)* event is specified to occur at the same time (*t2*) as its enabling event. Therefore the alternative **i** *(* Timeout *)*, in *State4*, is superfluous as it can never become enabled at time *t2* (this is clearly shown by the FSM).

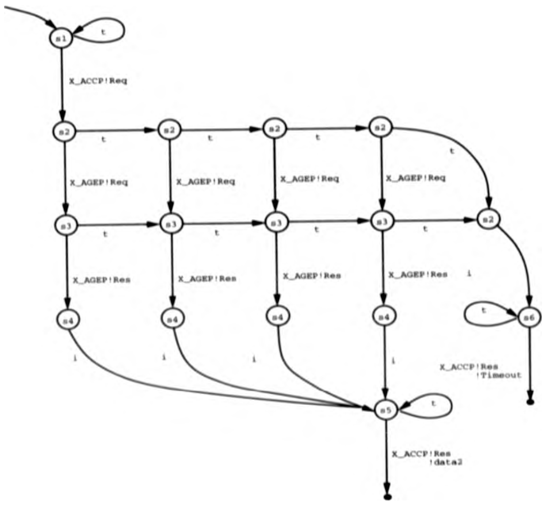


Figure 6.1: FSM of an X_Service_Agent

- Although this formal time-model allows us to prove and formally state certain quantitative timing properties (e.g. that event occurrences do occur at specified times), it lacks the expressive power needed to describe all the timing requirements of the X_Service and X_Service_Agent. We can express that events *must* occur by certain times, by suppressing the offer of *t*-events at appropriate times. However, we have no means to express the ASAP urgency of events, as asked for in the informal requirements. In the LOTOS description, we can only comment that X_AGEP events are to occur ASAP (see processes *State2* and *State3*).

- LOTOS descriptions in this *t*-event style are cumbersome to write, and not easily understood at a glance. Introducing yet more 'time-model mechanism' into descriptions, in order to express ASAP and other such timing constraints, makes descriptions almost incomprehensible.

6.2.4 Summary so far

We draw a few interesting conclusions from the above X_Service and X_Service_Agent examples:

- The informal time-model based LOTOS descriptions (in which timing constraints are expressed as LOTOS comments) provide no formal framework for reasoning about quantitative time-dependent behaviour. We cannot prove timing properties, nor can we unambiguously state timing constraints.

- The t-event based description is cumbersome to write, and not easily understood at a glance because of the large number of extra states introduced due to the explicit representation of the time mechanism. Moreover, without elaborate enhancement[2] (introducing an even greater number of states), this time-model is still inadequate in its support of timing features such as ASAP urgency, etc. This is an argument for developing a time-enhanced version of LOTOS in which timing constraints are easily expressed and understood by intuition.

- Questions may be raised concerning the nature and need for timing constraints such as: this event is to occur *as soon as possible*, *as late as possible*, etc. Such timing constraints are related to the ideas of *maximal progress* (a system should not idle if it can perform an action), *must timing* (specifying what actions a system must perform, and the times at which these actions occur), *multi-participant events* (events constrainted by more than one entity, e.g. observable events in LOTOS are constrained by both the system and its environment), etc. These, and other ideas will be investigated in the next section.

- In summary, we have seen that informal descriptions of timing concerns, even if expressed in otherwise formal descriptions, are inadequate; formal descriptions of timing concerns are cumbersome to write and difficult to understand unless the language includes an implicit model[3] for quantitative time.

6.3 Investigation of expressive power required for specifying timing concerns

In this section we investigate the expressive flexibility[4] required of a language for the specification of quantitative time constrained systems. We begin our investigative journey by examining the dependence on time of time predicates themselves. We then

[2]See section 6.6

[3]By "implicit" we mean that the framework for reasoning about quantitative timing concerns is built into the semantics of the language, as opposed to being expressed in the syntax of the description.

[4]We use this term to indicate that our concern lies with the *ease* of expression a language affords certain concepts, rather than the absolute expressive power of a language (e.g. relative to the power of a Turing Machine).

introduce a derivative of arc-timed Petri-Nets (PNs) and use this in our exploration of languages facilities needed for quantitative time-dependent specification.[8]

6.3.1 Past and future dependent time predicates

Consider the causally ordered events:

$$x \rightarrow y \rightarrow z$$

and the following description of their quantitative times of occurrence:

$$f(C(x)) = C(y) = f(C(z))$$

where $C(v)$ returns the quantitative time of occurrence of an event v, and f is a function whose domain and codomain are of a quantitative time sort.

Then $f(C(x)) = C(y)$ represents a *past dependent predicate*; the occurrence time of y is some function f of the occurrence time of x. Such a predicate (if true) constrains the occurrence time of an event y, relative to the occurrence time of an event x which is causally ordered before the event y.

$C(y) = f(C(z))$ represents a *future dependent predicate*. Such a predicate constrains the occurrence time of an event y, relative to the occurrence time of an event z which is causally ordered after the event y.

Future dependent predicates are of a declarative nature; e.g. in $C(y) = f(C(z))$, the value $C(y)$, at the occurrence of y, cannot be bound until the occurrence time of z is, later, established. While it is nice to have the expressive flexibility afforded by future dependent predicates, their dependence on a priori knowledge means that it may be difficult to simulate prototype descriptions which contain future dependent constraints. For example, consider a system described (in pseudo LOTOS syntax) as:

$$(x \rightarrow P[z] [] Q[z]) \wedge (C(x) = f(C(z)))$$

To decide if the system deadlocks when trying to execute an x event at time $C(x)$, we may have to explore both mutually exclusive paths of processes P and Q in order to establish possible values of $C(z)$). The problem of computing $C(x)$ is made worse if, say, $C(z)$ depends on $C(x)$ (incurring circular dependency). Also, future dependent predicates make it very easy for a designer to describe a system which is impossible to realize in real-life.

6.3.2 Initial ideas for time description

Petri-Nets are a nice medium in which to reason about simple distributed systems because of their clear graphical depiction of concurrent and of non-deterministic aspects. We introduce a derivative (which, for convenience, we call TPNs) of the arc-timed PNs

[8] We do not attempt to definitively state the set of quantitative time language facilities, but rather postulate some ideas and explore and evolve these. In this way we attain a *feel* for the topic.

of [Wal93, Bol90] as a tool for exploring language requirements for quantitative time-concerned specification. We gradually introduce properties for TPNs with a view to building a list of features we deem desirable for quantitative time-concerned specification.

We begin by stating the following properties of TPNs which capture what we initially consider as the basic features for time description.

P1. In TPNs *transitions* represent events.[6]

P2. An event is said to be *enabled* when all participants in the event are prepared to synchronize (and all their non-time related predicates are satisfiable[7]). This is represented if every input *place* to the event/transition contains a *token* and any predicates on these places are satisfied.

P3. An input arc may be labelled with a past-dependent time predicate.

P4. An event/transition is *fireable* when enabled, and if the conjunction of all the time predicates on its input arcs can be satisfied.

P5. An event/transition is *fired/occurs* when fireable, at a time which satisfies all of its time predicates. Unless, that is, the event is in the context of a choice expression, in which case one (and only one) of the set of enabled, mutually exclusive events (of the choice) will occur.

P6. A token is annotated with the firing time of the last transition which generated the token and for which a quantitative firing time can be established.

Figure 6.2 illustrates a TPN displaying the properties P1–P6. (In this section the identifiers $t1$, t_1, etc. represent time constants which are carried by tokens.)

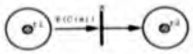


Figure 6.2: A typical TPN example

$$P(C(x)) \wedge (C(x) \geq t_1) \wedge (t_2 = C(x)) \tag{6.1}$$

Equation 6.1 explains the meaning of the TPN in figure 6.2. It says that, if the predicate P can be satisfied, then the occurrence time $C(x)$ of x will have a value greater than or equal to the token time t_1, and that token time t_2 will have the same value as $C(x)$.

[6] In the remainder of this section we freely interchange PN terms such as *transition* with LOTOS-like terms such as *event*, and vice versa.

[7] We consider time-related predicates as a separate issue.

Properties P1-P6 provide us with the following two important facilities for the specification of quantitative time concerns.

F1. The facility to specify that an event may occur only at constrained times (i.e. that time influences the occurrence of events).

F2. The facility to measure durations between events (not necessarily consecutive[8]).

These two facilities allow us to constrain events in quantitative time, relative to the quantitative occurrence time of other events.

6.3.3 Must timing

Property P5 allows the following facility.

F3. The facility to specify that if an event is enabled and its time predicate can be satisfied, then the event *must* occur. Unless, that is, the event sits within a set of mutually exclusive events which form the alternatives of a choice expression. Exactly one event of such a set *must* occur.

F3 indicates that our TPNs possess what several authors have called a *must* timing semantics. In the rival *may* timing semantics, events are not forced to occur if fireable. Obviously, for specification purposes, we will want to express the fact that certain events *must* occur at certain times in a system for it to be a correct implementation of the requirements. *May* timing does not give us the power needed to directly express facts such as these. For instance, if we simply wanted to specify that an event y is to occur within 2 units of time of x occurring, we might write the following TPN:

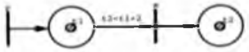


Figure 6.3: x within 2 time units of y

Without property P5, TPNs would revert to *may* timing semantics, and the specification in figure 6.3 would no longer ensure our requirement. This is because, without property P5, it is possible to choose a value for $t2$ (say ($t2 \geq (t_1 + 1)$)) which does not satisfy the predicate ($t_2 < t_1 + 2$). For this TPN, the choice of such a value would, in effect, result in a deadlock situation where the event x would never occur.

[8]Using TPNs this implies extending token annotation so that tokens also carry a history of the occurrence times of the transitions which they have passed through (i.e. the transitions responsible for (re-)generating them).

Another illustration of the inadequacy of *may* timing (for specifying a symmetric time-out mechanism) can be found in [Bol90, BL91].

We can use *must* timing to express *may* timing, although not vice versa. For example, leaving property P6 in place, but changing the predicate in figure 6.3 from $(t_2 < t_1 + 2)$ to $(t_2 \geq t_1)$, we express that x *may* occur within 2 time units of y.

Must timing within the context of a choice expression forces exactly one of the enabled alternatives of the choice to occur. Thus, given the TPN of figure 6.4, exactly one of the events x or y *must* occur since they are both fireable. Event z is not firable as its time-predicate is not satisfiable (since the token time $t_1 > 3$).

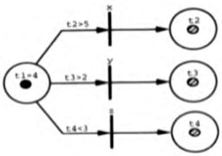


Figure 6.4: *Must* timing within a choice context

6.3.4 Compositionality of time predicates

Properties P2–P5 tell us that the time constraints for an event are formed from a conjunction of the time constraints which each input arc imposes on the event occurrence. This gives us the power to build global time constraints out of local ones — the power to (de)compose separate timing concerns. We restate this as:

F4. The facility to express local time predicates.

F5. The facility to compose local time predicates.

Figure 6.5 illustrates a TPN in which local time predicates (P and Q) are expressed and composed.

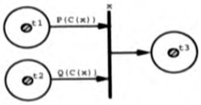


Figure 6.5: Example Composition of Local Time Predicates

$$P(C(x)) \wedge Q(C(x)) \wedge (C(x) \geq t_1) \wedge (C(x) \geq t_2) \wedge (t_3 = C(x)) \tag{6.2}$$

Equation 6.2 (which reflects the meaning of the TPN in figure 6.5) says that, if the predicates P and Q can be satisfied, then the occurrence time $C(x)$ of x will have a value greater than or equal to either of the the token times t_1 and t_2, and that token time t_3 will have the same value as $C(x)$.

6.3.5 Relative ordering

We introduce a new TPN property:

P7. A transition whose input arcs do not have any associated time predicates, will occur if enabled. Its (quantitative) firing time is not established.

This provides:

F6. The facility to express relative ordering constraints over events (without quantitative timing constraints).

Figure 6.6 illustrates property P7.

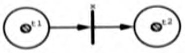


Figure 6.6: Example of the use of relative ordering

$$(C(x) \geq t_1) \wedge (C(x) \leq t_2) \wedge (t_1 \geq t_2) \tag{6.3}$$

Equation 6.3 says that event x occurs at a time $C(x)$ in the interval delimited by the two token times t_1 and t_2, and token time t_2 is greater than or equal to t_1. Notice that token time t_2 does not reflect the time $C(x)$, and the occurrence time $C(x)$ of x

is not recorded in the history-annotation of this token. Thus, in effect, we have not established a quantitative occurrence time for x.

Using facility F6 we can specify that, for example, an event must occur if enabled, and that it can occur at any time provided that this time does not preclude the occurrence of the directly succeeding event. We consider the following two TPNs to clarify this example.

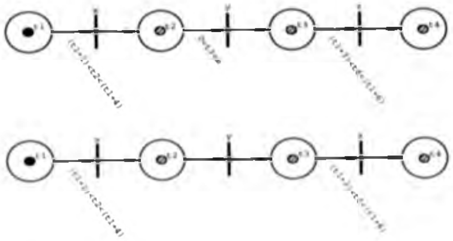


Figure 6.7: Example with quantitative time, then relative ordering

A legitimate trace (in which events are annotated with their quantitative occurrence times, e.g. x_3 means 'x at time 3') for the first TPN (figure 6.7) is:

$x_3 \rightarrow y_6 \rightarrow$ ***deadlock***

However, this is not a legitimate trace for the second TPN (figure 6.7). In the second TPN the occurrence of time of y is not established because of property P7 and so the occurrence of y does not preclude the occurrence of z (and so a ***deadlock*** situation is avoided).

6.3.6 Environment interactions

P8. A dotted input arc for a transition indicates that the environment is a participant in this transition. Such a transition cannot be said to be enabled until the places, both those controlled by the environment and those controlled by the system, each contain a token. The environment can constrain the occurrence time of events through its own predicates.

Property P8 recognizes that observable events require the participation of the environment; until the environment agrees to participate in the event, the event cannot be enabled and so cannot be forced to occur. We restate this as:

F7. The facility to distinguish between internal events and events in which the environment is a participant.

The TPN in figure 6.8 possess property P8.

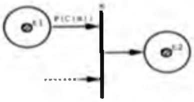


Figure 6.8: Environment interaction example

$$P(C(x)) \wedge (C(x) \geq t_1) \wedge (t_2 = C(x)) \wedge (env\_constraints\_on(C(x))) \qquad (6.4)$$

Equation 6.4 (which reflects the meaning of the above TPN) is the same as equation 6.1 with a placeholder for environment constraints on $C(x)$.

6.3.7 An overview of clocked models

Models in which local clocks co-exist with behaviour Equation 6.2 leads us to examine the concepts of local/global time. Equation 6.2 is actually just one of Lamport's "Logical Clock Implementation Rules" [Lam78]: local clocks synchronize their values when the entities they influence synchronize on an event.

Until now, our TPN properties have implemented what are essentially local clocks, each of which co-exists with an independent (sequential) stream of behaviour. This is because each token, travelling along a sequential stream of transitions, carries with it and maintains what amounts to the value of a local clock. Figure 6.9 should help clarify this notion.

Each shaded area marks the existence and extent of influence of a local clock. All transitions within one shaded area perpetuate and are influenced by the local clock for that area. Where a transition (e.g. x) falls under the influence of more than one local clock (diagrammatically, more than one shaded area overlaps a transition), the local clocks (e.g. C_1 and C_2) synchronize their times and a set of new local clocks is generated (e.g. C_3, C_4) whose initial values are that of the synchronized clocks (C_1 and C_2).

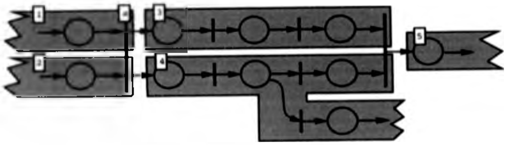


Figure 6.9: Existence of local clocks in a TPN

Clocks used as description tools In the real world, a global clock time is not a concept that is readily available. Distributed system implementations may use a distributed set of synchronizing logical clocks or, better still, synchronizing physical clocks to establish a satisfactory total ordering (in quantitative time) of events in the system.

Ideally, a specification language should provide a designer with a set of clocks built into the language. Clocks in this set may run at different rates, may synchronize, and may be used to influence the outcome of time predicates. This provides the flexibility needed to support:

- problem domain descriptions (high level, abstract specifications) which use only one clock (a global clock time)
- solution-domain descriptions (low-level specifications) which use a distributed set of synchronizing local clocks.

If global time is used as a description or design device, the implementation may still be based on local clocks if it maintains the logical clock properties of the original description. Moreover, the use of either global or local time in a design description may or may not imply the assumption of (the support of) either global or local clocks. If a description contains time constraints this does not necessarily require the implementation to have physical clocks — time in a specification may serve as an abstract means of stating time-dependent behaviour and performance requirements (something which may be forced onto the implementation when it is placed in context).

We summarize that a language for describing quantitative time-dependent distributed systems should have:

F8. The facility to declare a set of clocks. Clock support mechanisms are implicit (built into to the language). A clock may influence a set of time predicates. Clocks may run at different rates and may synchronize their time values.

Orthogonally-clocked models Facility F8 requires clocks to exist independently of behaviour carriers (i.e. transitions, arcs, places, etc.). (This is in contrast to the model shown in figure 6.9.) This is because F8 allows clock constraints to be applied orthogonally to sequential behaviour structure — i.e. each clock may be used to constrain events which occur in different causally independent 'behaviour streams'. The example in figure 6.10 may help clarify this: three clocks exist independently of the tokens, places, transitions and arcs; they constrain, orthogonally to the 'behaviour streams', the occurrence times of transitions.

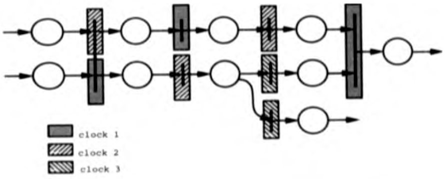


Figure 6.10: An orthogonally-clocked model

Enhancing TPNs to support facility F8 requires that we make a few radical changes to our previous TPN properties. Tokens are no longer responsible for carrying around 'local clocks'. Instead:

P9. A system of clocks $C_1, \ldots, C_n$ exists independently of the TPN behaviour carriers that we have seen so far (i.e. tokens, places, transitions, arcs). If a predicate P can be satisfied by a time value $C_i(x)$, where $C_i(x) \geq C_i(present)$, then clock C_i will reset its *present* value $C_i(present)$ to equal $C_i(x)$.

A token is annotated with a chronological history of the occurrence times, each relative to some clock in $C_1, \ldots, C_n$ of the transitions which have (re-)generated it.[9]

Now consider figure 6.11 which puts property P9 into practice.

[9] We are only interested in exploring the range of expressive flexibility needed to describe quantitative timing concerns, not providing a definitive treatment of TPNs, hence our vague treatment of issues such as how and where clocks are initialised, how token histories are merged, etc.

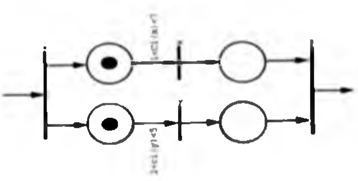


Figure 6.11: An orthogonally-clocked example

In the above TPN, clock C_i applies timing constraints orthogonally to behaviour structure.

6.3.8 Must timing in the context of parallel expressions

Although we have gained greater expressive flexibility with the introduction of orthogonal clocks, we have done so at the expense of increasing the complexity of enforcing our *must* timing regime of facility F3.[10] In a parallel expression context, *must* timing has to ensure the sensible interleaving of event sequences[11].

Consider the following three (not necessarily legitimate) time annotated traces (interleavings) of the TPN in figure 6.11.

$$w \rightarrow x_{2_1} \rightarrow y_{3_1} \rightarrow z \tag{6.5}$$

$$w \rightarrow y_{4_1} \rightarrow x_{6_1} \rightarrow z \tag{6.6}$$

$$w \rightarrow x_{5_1} \rightarrow \textit{deadlock} \tag{6.7}$$

Traces 6.5 and 6.6 are legitimate deadlock free traces, while trace 6.7 is an illegitimate trace (by property P5) which deadlocks after the event x occurs at time 5_1. The *must* timing semantics must disallow trace 6.7 in support of property P5 which dictates that, since both events x and y are enabled (outside the context of choice expressions) and their fireable times are satisfiable, both events *must* occur.

The following traces represent legitimate, sensible *must* timed interleavings (maximal traces) of the TPN in figure 6.11:

$$\begin{aligned}
\{&w \rightarrow x_{2_1} \rightarrow y_{3_1} \rightarrow z, w \rightarrow x_{3_1} \rightarrow y_{3_1} \rightarrow z, w \rightarrow y_{3_1} \rightarrow x_{3_1} \rightarrow z,\\
&w \rightarrow y_{3_1} \rightarrow x_{6_1} \rightarrow z, w \rightarrow y_{3_1} \rightarrow x_{6_1} \rightarrow z, w \rightarrow x_{1_1} \rightarrow y_{4_1} \rightarrow z,\\
&w \rightarrow x_{2_1} \rightarrow y_{4_1} \rightarrow z, w \rightarrow x_{3_1} \rightarrow y_{4_1} \rightarrow z, w \rightarrow x_{4_1} \rightarrow y_{4_1} \rightarrow z,\\
&w \rightarrow y_{4_1} \rightarrow x_{4_1} \rightarrow z, w \rightarrow y_{4_1} \rightarrow x_{6_1} \rightarrow z, w \rightarrow y_{4_1} \rightarrow x_{6_1} \rightarrow z\}
\end{aligned}$$

[10] Gaining expressive flexibility at the expense of computability is a problem which pervades language design.

[11] This is akin to the notion of [QAF90] on well-formed interleavings.

Sensible, *must* timed interleaving[12] is desirable, but in order to enforce it we must be able to ascertain particular information from time predicates.

Within time predicates, we use functions for generating sets of time values, i.e. generator functions. We describe any possible generator function as:

$$GenFunc : TimeSort, \ldots, TimeSort \rightarrow TimeSetSort$$

To enforce *must* timing we require information from two auxiliary functions that we will call ***IsIn*** and ***isGTAllMembersOf***.

$$IsIn : TimeSort, TimeSetSort \rightarrow BooleanSort$$
$$isGTAllMembersOf : TimeSort, TimeSetSort \rightarrow BooleanSort$$

We require that generator functions be restricted such that the auxiliary functions ***IsIn*** and ***isGTAllMembersOf*** are defined for all generator functions *GenFunc*.

To explain the rôles played by the functions ***IsIn*** and ***isGTAllMembersOf*** in the enforcement of *must* timing, consider the following scenario.

- Given a set of concurrent, enabled events $\{a_1, \ldots, a_n\}$[13]
- whose occurrence times are constrained to the time values generated by the functions $GenFunc_1, \ldots, GenFunc_n$ (respectively)
- then, to enforce *must* timing for these events (i.e. to ensure that each of the events $a_1, \ldots, a_n$ are fired)
- we define the set of (firings) transitions for these events to be: $-a_i t_i \rightarrow$, $1 \leq i \leq n$, where:
  - t_i ***IsIn*** $GenFunc_i$
    i.e. the firing time t_i, of event a_i, is a value within the set of values generated by its generator function $GenFunc_i$
  - *not*(t_i ***isGTAllMembersOf*** $GenFunc_j$), for $1 \leq j \leq n$
    i.e. that an event a_i does not pre-empt the occurrence of any other event. We ensure that the firing of event a_i at time t_i will not make obsolete all of the firing times of each one of the other enabled events.

isGTAllMembersOf is used to ascertain whether an event can occur in the future. For example, say event y is constrained to occur only at times generated by the generator function *GenFunc1*, and say the present time is t. Then to test whether event y can possibly occur in the future, we use t ***isGTAllMembersOf*** *GenFunc1*. The answer ***False*** would indicate that event y may occur in the future. While the answer ***True*** would indicates that y can never occur in the future — i.e. we have ***deadlock*** with respect to event y; all the times generated by *GenFunc1* are obsolete at time t.

[12]Hereafter, simply called '*must timing*'

[13]Where no a_i, $1 \leq i \leq n$, is a member of a set of mutually exclusive alternative events, i.e. a_i is not combined as $a_i [] P$

6.3.9 Time policies

We have seen how, with the support of *must* timing semantics, the specifier can express that a certain event *must* occur *somewhere* within a certain *time window*. We consider this to be the *Normal time policy*. The Normal *time policy* does not influence the *position*, within the time window, an event occurs.

In this subsection we consider two new *time policies*. Each new *time policy* influences the *position* within a time window (specified by a time predicate such as $x < t < y$), an event occurs.

These two new *time policies* are *ASAP* ('as soon as possible') and *ALAP* ('as late as possible').

Bolognesi *et al.* [BL91] defines an ASAP transition to be: a transition which is fired as soon as it becomes fireable. As a complement to the ASAP notion, we propose ALAP. We define an ALAP transition to be: a transition which is fired at a time after which it would never again be fireable. Therefore, the ASAP *time policy* influences the event to occur at the start of its time window; whereas the ALAP *time policy* influences the event to occur at the end of its time window.

We say that time policies "influence" rather than dictate, since time policies (like time predicates) may be composed, and the resulting conjunction of dissimilar time policies may not reflect every individual time policy. In this case, an individual time policy merely "influences" the policy resulting from the conjunction. The actual result of combining dissimilar time policies is defined by the semantics (see figure 6.16 in section 6.5.4.1). These semantics state that combinations involving only the one type of time policy will result in that time policy. Combinations involving only Normal and ASAP will result in ASAP. Combinations involving only Normal and ALAP will result in ALAP. Combinations which involve both ASAP and ALAP will result in Normal (i.e. at least one ASAP will annihilate any number of ALAPs, and vice versa, to result in Normal).

We can think of time policies other than Normal, ASAP and ALAP. However, in this thesis we content ourselves with only these. Also, we can think of alternative results for the combinations of dissimilar time policies. For example, we could define the result of any combination of time policies to be the time policy which occurs most frequently in the combination. We have chosen our particular semantics for time policy combination on intuitive grounds. (Section 6.5.4.1 says more on the subject of time policy semantics.)

By default, time predicates are associated with the Normal time policy. Alternatively we can associate a time predicate with an ASAP or ALAP time policy.

The following example illustrates the effects of time policy combinations. Consider the three TPN contexts shown in figure 6.12.

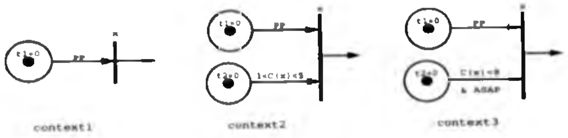


Figure 6.12: Example contexts illustrating time policies

The following table describes the effects of the various combinations of time predicates and time policies in each of the three contexts in figure 6.12, on the sets of possible occurrence times of event x.

time predicate & policy (PP)	occurrence times of event x		
	context 1	context 2	context 3
$PP = 1 \leq C(x) \leq 6$	$C(x) \in \{1,2,3,4,5,6\}$	$C(x) \in \{2,3,4\}$	$C(x) \in \{1\}$
$PP = ASAP$	$C(x) \in \{ 0 \}$	$C(x) \in \{2\}$	$C(x) \in \{0\}$
$PP = (1 \leq C(x) \leq 6) \wedge ASAP$	$C(x) \in \{1\}$	$C(x) \in \{2\}$	$C(x) \in \{1\}$
$PP = ALAP$	$C(x) \in \{ \mathit{infinity} \}$	$C(x) \in \{4\}$	$C(x) \in \{0,1,2,3,4,5,6,7\}$
$PP = (1 \leq C(x) \leq 6) \wedge ALAP$	$C(x) \in \{6\}$	$C(x) \in \{4\}$	$C(x) \in \{1,2,3,4,5,6\}$

To ensure that the use of ASAP and ALAP time policies does not render a specification unexecutable, we insist that the three auxiliary functions *isUpperLimited*, *Min* and *Max* are definable over the sets of time values produced by generator functions. The function *isUpperLimited* is used to establish if a given set of generated time values has an upper limit, and hence if there is an ALAP ('as late as possible') time value with the given set of time values. If a given set of time values does have an upper limit, then *Max* can be used to return the maximum (upper limit) value. The function *Min* is used to return the minimum value in a given set of time values, i.e. the ASAP ('as soon as possible') value. Note that it is not necessary to do an *isLowerLimited* check before applying *Min* to a set of time values, because all sets of time values have at least 0 as their lower limit.

Sections 6.5.1 and 6.5.2 incorporate and define the auxiliary functions *IsIn*, *isGTAllMembersOf*, *isUpperLimited*, *Min* and *Max* in the definition of a time-extended version of LOTOS.

6.3.10 Limitations on enforcing must timing

We have seen examples of how *must* timing can support the sensible interleaving of the two causally independent[14] behaviours (section 6.3.8 figure 6.11). Can *must* timing support sensible interleaving for more complex behaviours? Consider the two causally independent behaviours in figure 6.13.

[14]ignoring clock communication

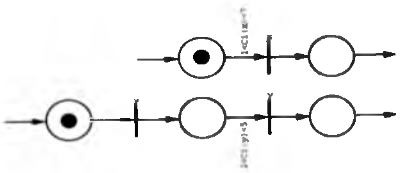


Figure 6.13: Two causally independent behaviours

We would like *must* timing to ensure that the trace:

$x_6 \rightarrow$ ***deadlock***

is not legitimate. However, this is not as simple as the case shown in figure 6.11. In figure 6.11 we use *must timing* to ensure a sensible interleaving between any set of *enabled* events. *Must* timing does not necessarily extend this sensible interleaving property to interleaved sequences of events. If we consider *must* timing as a function over behaviour expressions, we can re-state what we have just said as:

$$MUST(a; P|||b; Q) \Rightarrow SensiblyInterleaved(a, b) \qquad (6.8)$$

$$MUST(a; P|||b; Q) \not\Rightarrow SensiblyInterleaved((a; P), (b; Q)) \qquad (6.9)$$

The behaviour in figure 6.13, is an instance of equation 6.9. Event y is not initially enabled, and we cannot ensure that x will occur at a time (relative to clock $c1$) such that y may still occur.

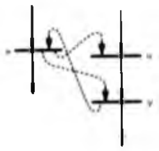


Figure 6.14: A network of possible causal relations

Replacing the $\not\Rightarrow$ with $\Rightarrow$ in equation 6.9, would imply that the *MUST* function has the ability to compute sensible interleavings from a network of possible causal relations, formed from both *action-prefix* operators and the orthogonal clock constraints. Figure 6.14 shows the network of possible causal relations for figure 6.13, where the solid arrows denote causal relations due to *action-prefix* operators, and the broken arrows denote causal relations due to clock constraints.

From figure 6.14, we can see that forming a sensible interleaving of the concurrent behaviours of figure 6.13 requires a priori knowledge of how the behaviours unfold. We can only form a sensible interleaving after we have established all the possible causal relations (see equation 6.10).

$$SensiblyInterleave(UnfoldBehaviour(a; P|[]|b; Q)) \tag{6.10}$$

With respect to LOTOS specifications, equation 6.10 implies that we would have to "simulate" a LOTOS specification in two "passes". This has all kinds of repercussions for the definitions of legitimate behaviour, conformance, etc.

Is there any means of producing sensible interleaving of concurrent event-sequences in one pass? We could take the view that our problem stems from attempting to produce a sensible linear trace from concurrent behaviours. Therefore, we might consider revising the use of simple linear traces as a "normal form", and instead use more complex structures such as "labelled, partially ordered multi sets", "augmented traces", etc. [CdC91, Fid92]. The function *UnfoldBehaviour*, from equation 6.10, results in a normal form of one of these more complex types. However, these normal forms are quite different from the simple linear traces of standard LOTOS.

Alternatively we can insist that, for a sensible interleaving of any two concurrent event sequences, all event offers must have explicit constraints for each of the clocks which are visible to both concurrent event-sequences. (What we advocate contradicts, in the situations mentioned in the previous sentence, our reason for wanting the relative ordering facility F6.) So, for instance, if we want sensible interleaving of the behaviours in figure 6.13, we must attach an explicit constraint, referencing the clock c_1, to the event a.

6.3.11 From TPNs to LOTOS

TPNs have served well as a tool for exploring language requirements for quantitative time concerned specification. However, we now move from TPNs to LOTOS. TPNs are useful for simple descriptions, but lack the powerful descriptive features of LOTOS (e.g. recursion, parameterization, definition and instantiation, etc.) which allow the compact expression of complex and infinite behaviours.

6.3.12 Simulation of physical clocks

In subsection 6.3.7 we suggested the idea of a set of orthogonal clocks, in which clocks may run at different rates. In subsection 6.3.10 we concluded that each event, in a concurrent sequence, must explicitly reference each visible clock, for *must* timing to produce sensible interleavings. Referencing all visible clocks could become cumbersome if the set of visible clocks, for a concurrent sequence, is large. This leads us to re-examine the need for having more than one clock.

The idea of introducing clocks into LOTOS is to capture quantitative time properties of distributed systems. In essence, clocks provide a means of communication between (otherwise) independent processes. Logical clocks (as discussed in subsection 6.3.7) do not provide any extra means of communication over and above that of event synchronization. What is more, [Lam78] shows how logical clocks may be implemented using event synchronization as a basis — the same conclusion drawn in subsection 6.3.7. Therefore, purely logical clocks (as in subsection 6.3.7) would not increase the expressiveness of LOTOS. Moreover, [Lam78] explains two possible reasons why total ordering using logical clocks may not be completely satisfactory.

In [Lam78] Lamport defines a system of physical clocks suitable for our purposes. He states a number of conditions that physical clocks must satisfy for them to be useful for the purpose of establishing total event ordering. Basically, physical clocks should run continuously at approximately correct rates. Knowing some information on transmission delay limits, clock rate limits, etc., time-stamped messages can be used to synchronize physical clocks to within known limits. Implementation constraints allowing, it may then be possible to establish a completely satisfactory total ordering of events in a system.

Lamport's physical clocks are analogous to our orthogonal clocks of subsection 6.3.7, in that they provide a *special* means of communication, separate from that of event synchronization, between (otherwise) independent processes. In reality the *special*, instantaneous communication afforded by physical clocks is not communication in the usual sense ("message communication"), but represents the sharing of a common property by which physical clocks measure time. (The common property may be, for example, the universally constant vibration rate of a quartz crystal.)

Hence, introducing physical clocks (i.e. orthogonal clocks) into LOTOS would allow us to describe, in a direct way, quantitative timing concerns of systems which rely on the *special* means of communication afforded by physical clocks. The ordinary event synchronization of LOTOS is left to model normal "message" communication.

Since the essence of physical clocks is that they enjoy a common property which governs their measurement of time, the crux of incorporating physical clocks into LOTOS is to provide a mechanism for ensuring/dispensing the basic tick rate of physical clocks. We can embody this mechanism as the *base clock*. The idea of a set of clocks can be modelled by stating time constraints as some offset of the base clock. Then, to ensure sensible interleavings of concurrent event sequences (e.g. figure 6.13), we need only ensure that each event explicitly reference this one *base clock*.

6.4 TLOTOS in comparison with existing work

Having explored and evolved a set of features which we believe to be useful for the expression of quantitative time concerns, and before proceeding to incorporate these features into a time-extended version of LOTOS, we use this point in our chapter to summarize our set of features and contrast these with existing work in the field.

6.4.1 Summary of quantitative time features for extending LOTOS

The following list of features have been distilled from the investigation in the previous sections of this chapter. We incorporate these features in a time-extended derivative of LOTOS known as TLOTOS.

1. TLOTOS supports the notion of physical clocks. TLOTOS realizes the notion of physical clocks by having a *global*, built-in, *base clock*. All TLOTOS *processes* may access this base clock to establish a universal value which represents an absolute measure of the time which has passed in the system at any particular instant.
2. The facility to specify that an event may occur only at constrained times (i.e. that time influences the occurrence of events).
3. The facility to measure durations between events (not necessarily consecutive).
4. The facility to express local quantitative time constraints and to compose these, in a similar fashion to the composition of local *selection-predicates* in LOTOS.
5. The facility to be compatible with LOTOS's relative ordering feature. A TLOTOS description with no quantitative time constraints will have the same semantics as the syntactically identical standard LOTOS description.
6. The facility to support the expression of relative quantitative time constraints. (This allows us to simulate the effects of clocks which run at different rates.)
7. The facility to support "must" timing, in the sense that any event (that is not an alternative in a choice expression and) that is fireable will occur.

   Fireable events (which are not direct alternatives in a choice expressions) are forced to occur within their allotted time windows (specified by *time-offers*). TLOTOS semantics ensure sensibly interleaved event sequences.
8. The facility for *time-policies* which operate in association with *time-offers* or by themselves. TLOTOS offers three *time-policies*: Normal (the default); ASAP which influences an event to occur *as soon as possible* (similar to "urgency" [BL91] or "maximal progress"); and ALAP which influences an event to occur *as late as possible*. (In essence, ASAP gives ordinary events priority over events which represent the passing of time, while ALAP gives the inverse priority.)

6.4.2 Comparisons with existing work

A variety of solutions for extending LOTOS with quantitative time have already been proposed by other authors. The following paragraphs very briefly summarize points of interest in these solutions, and contrast these solutions with our own TLOTOS solution.

6.4.2.1 "TIC: A timed calculus for LOTOS"

Of existing proposals, [QAF90]'s TIC is the most similar to our TLOTOS. In TIC, an *action-denotation* has a set of of possible occurrence times. Values of this set are interpreted as offset times from the occurrence of the event which directly causes the event in question. Although a global clock is implicitly defined by the semantics of TIC, this clock cannot be accessed at the syntax level. There is no facility for establishing a value for the occurrence time of an event. Hence it is not possible to measure the duration between events, nor is it possible to express absolute timing constraints or timing constraints which are relative to anything but the occurrence time of the directly 'causing event'.

TIC enforces "must" timing, and TIC interleaving generates only sensible mergings of event sequences (in the sense discussed in section 6.3.8). TIC semantics for *parallel-expressions* are very similar to semantics for TLOTOS *parallel-expressions* (see section 6.5.4.3). TIC applies the auxiliary operator *Old* to denote the aging of inactive behaviours. This is necessary because, all TIC behaviours carry a value of the time[15], whereas in TLOTOS individual behaviour expressions do not carry a value of the time, but access an explicitly defined global clock (value).

TIC offers no facility for specifying that an event is to occur ASAP or ALAP. For example, consider the following attempt to emulate ASAP in TIC. We want to specify that event a is to occur ASAP. In the particular case:

$i;\ a\ 0$

event a is specified to occur zero time after its directly causing event. If this *action-prefix* expression is considered in isolation, this does cause event a to occur ASAP. However, if we consider this TIC *action-prefix* expression in a *parallel-expression* context:

$b;\ a\ 0 \ldots |[a]| c;\ a\ 0 \ldots$

we are no longer guaranteed to achieve the ASAP effect. If the quantitative occurrence times of events b and c are not identical, then event a cannot occur. Therefore, this simple approach to emulating ASAP in TIC does not, in general, work. Emulating ASAP (or ALAP) in TIC would require the construction of a much more elaborate mechanism (e.g. see section 6.6).

Like TLOTOS, TIC separates, within *action-denotations*, the *time-offer* from the *selection-predicate*. This is necessary for the definition of "must" timing, which requires that functions such as *IsIn* and *GTAnymember* be definable over a *time-offer* (see sections 6.3.8 and 6.5.4, and appendix C). If the *time-offer* was incorporated into the *selection-predicate* (as is the case for [vHTZ90]'s CELOTOS, see below), it would then be impossible to apply an ordering function, such as *GTAnymember*, to the *Boolean* result of the *selection-predicate*.

6.4.2.2 "CELOTOS: LOTOS extended with clocks"

[vHTZ90]'s CELOTOS maintains sets of explicit clocks which may be started and read by CELOTOS behaviour expressions. Independent expressions[16] may share the same set of clocks. Testing clock values within *selection-predicates* allows time to influence the occurrence of events. This, combined with starting clocks within special syntactic constructs, facilitates the measurement of durations between events (not necessarily

[15]When considered at a global level, these values of time define a global clock

[16]independent, apart from their common notion of the (quantitative) time

consecutive).

TLOTOS is very similar to CELOTOS with respect to the afore mentioned features, except that in TLOTOS only one global clock is (semantically) maintained. For TLOTOS, we take the view that the global clock emulates the special means of communication afforded by the universal constant found in real-life physical clocks. We leave it to the user to define, if necessary, other clocks which may possibly run at some function of the base clock rate (thus, for example, representing the timing defects in imperfect physical clocks).

CELOTOS does not support "must" timing. Thus, although the following CELOTOS behaviour expressions P and Q **exit**, when considered separately:

> **P :=i 3; exit**
> **Q :=i 4; exit**

when considered as $P|||Q$ they may not both exit.[17] This is undesirable given that $|||$ is supposed to indicate that they are causally independent.

There are no facilities in CELOTOS for supporting ASAP or ALAP facilities.

6.4.2.3 Urgent and timed interactions

[BL91]'s U-LOTOS and T-LOTOS are defined on the basis of "urgent-interactions" (related to the idea of "maximal progress"). In U-LOTOS, an urgent-interaction is guaranteed to occur ("must" occur) ASAP (i.e. at its enable-time). T-LOTOS is more flexible, guaranteeing that a timed interaction occurs within a specified time window. Such a time window is defined to range from the enable-time of an interaction event until some specified time.

U-LOTOS and T-LOTOS semantics consist of a set of inference rules for deriving action transitions, a set of inference rules (orthogonal to those for actions) for deriving aging transitions, and auxiliary functions for establishing the times at which action transitions are possible. The two sets of inference rules realize, at a semantic level, that time passing events and action events (which take no time to execute) are separate. This contrasts with TLOTOS, in which all transitions are annotated with both time and action attributes. Bolognesi *et al.* claim that having separate time transitions and action transitions make it simpler to express "maximal progress" timing (or, as Bolognesi *et al.* term it, "action necessity"). We fail to see the claimed advantage — in our opinion, T-LOTOS's and TLOTOS's schemes are equal for the expression of action necessity. Indeed, we argue that it is more intuitive to think of dual 'time-action' transitions, where actions *must* occur at the specified times if time is to progress.[18]

With separate time and action transitions, U-LOTOS and T-LOTOS semantics must use auxiliary functions (α_a in U-LOTOS, and age_a in T-LOTOS) to check time-transitions, and then use this information to *force* action transitions to occur when necessary. The definition of these auxiliary functions is not trivial. They are defined in a denotational style, and are required to check all inference rule derivations from any one state. To ensure that this checking process does not diverge, U-LOTOS

[17] the i at time 4 may occur, pre-empting the i at time 3.

[18] The reader may need to read section 6.5.4 before fully understanding this discussion

and T-LOTOS disallow non-action guarded recursion.

By employing composite 'time-action' transitions, TLOTOS has no need of such auxiliary functions, and consequently does not have to restrict recursion.

Although T-LOTOS supports "must" timing, and supports the notion of ASAP events, it does not facilitate the measurement of inter-event duration, nor the expression of ALAP, and is restricted in its ability to express local time constraints.

In T-LOTOS we can locally express time delays using a delay prefix operator: "wait for t units of time and then behave like B". The effect of a local time delay operator, used within a behaviour expression, is felt in all parts of the system which are composed in parallel with the behaviour expression. However, it is not possible to locally express the fact that an observable event must occur within a particular time window. This is because T-LOTOS uses its '*timer* $a(t1, t2)$ *in* B' construct to both declare the gate a, and to indicate that events at gate a must occur within the time interval specified. (It is only possible to declare *must* timing constraints over a set of gates when actually declaring the gates — i.e. it is only possible to state "urgency" requirements on internal events.) The following example illustrates this. In section 6.5.1, we will see that in TLOTOS we can write (in abbreviated syntax):

> P[a]...
> **where** P[a] := a ASAP;...

to specify that event a must[19] occur ASAP. Notice that the ASAP constraint on event a is local to P. The most similar specification in T-LOTOS is:

> **timer** a(0,0) in P[a]...
> **where** P[a] := a;...

In the T-LOTOS specification, we could not locally specify the timing constraint for the event a within process P — the **timer** statement is not local to P. In T-LOTOS such timing constraints can be applied only to internal events.

6.4.2.4 "Simulating real-time behaviour"

Fidge in his paper [Fid90] extends Basic LOTOS by giving "events" durations. His "real-time simulator" produces a trace in which each 'time consuming' event is denoted by delimiting start- and stop-labels. These start- and stop-labels are attributed with time values, such that their difference is equal to the duration of the "time consuming" event. Fidge introduces "concurrency" operators which interleave sequences of "time consuming events" such that the two time intervals of any two events from the concurrent sequences do not overlap. Fidge has the normal LOTOS *parallel-operator* produce an interleaving in which events (time intervals) may overlap, indicating the truly parallel occurrence of these "time consuming" events.

Within a Fidge specification, there is no means of establishing the start or stop times of events. The concept of "must" timing is not applicable as there is no means of specifying that events are to occur at particular quantitative times (although we know that an event starts at a time equal to the sum of the durations of the events ordered

[19]with the exception that, if the a *ASAP* offer synchronises with an a *ALAP* offer, then the ASAP urgency will be annihilated, see section 6.5.4.1

causally before). All events start as soon as they possibly can, unless an explicit "delay" operator is used. Also, Fidge suggests attributing LOTOS operators with durations, in a way similar to that in [ERP90] (see section 6.4.2.6).

6.4.2.5 Compound and t-events in LOTOS

[AQ90] introduces the idea of "compound events" ("c-events"). Two or more "simple events" ("s-events") may be combined using the new "*" operator to indicate the simultaneous occurrence of these events.

[AQ90] builds on the compound event concept, describing how a c-event may contain the special s-event "t" (t-event). A t-event occurrence represents that one unit of time has passed. The authors suggest that their concept of c-events containing t-events avoids the cumbersomeness of purely interleaved semantic interpretations of 'a set of events which occur at the same time'. Traces of this calculus consist of a linear chain of causal relations and c events. When each c event is considered as a set of s-events, traces could be considered as being similar to "partially ordered multi-sets", etc. [CdC91].

This c-event, t-event calculus is quite 'low-level', in that t-events appear at the syntactic level . No facilities are provided for specifying event occurrence times in terms of a time metric, establishing event occurrence times, specifying ASAP or ALAP concerns, etc.

[AQ90] introduces questions on the subjects of "divergence and realism". "Divergence arises when a loop of internal events occurs within a finite time interval." This may violate the "realism requirement" that an infinite number of events may not occur within a finite period of time. This implies that a time-extended LOTOS description may be wrong if "the total time consumed (in the real system) by events occurring within a time interval becomes of the same order of magnitude as the time interval itself". Thus we must be careful if we either construct recursive loops in which events may occur immediately (i.e. with no time constraints), or have unbounded creation of processes in parallel (see [QAF90]). For example, in:

```
    x{setEQ(2)}; (* x occurs at time 2 *)
    P[y](VeryLargeNum) >>
    z{setEQ(3)}; (* z occurs at time 3 *)

where
    process P[y](n:Nat) exit :=
            [n gt 0] -> y; P[y](n-1)
        []
            [n eq 0] -> exit
    endproc (* P *)
```

we should consider whether or not the *VeryLargeNum* of occurrences of event *y* can in fact *realistically* happen within the 1 unit time interval between events *x* and *z*. (As specifiers, we may parry this problem by labelling it as an implementation concern.) These issues relate to the notion of zero separation between events as an approximation of negligible duration, which is in itself a controversial notion.

6.4.2.6 LOTOS operators with durations

[ERP90] introduce time durations for LOTOS operators, including ACT ONE function evaluation. Their work emphasizes concerns about resource performance with respect to implemented LOTOS specifications. Thus, for example, inter process communication delays in multi-processor systems are modelled by imposing time durations for process synchronization. Similarly, system interrupts, context swaps, etc. can be modelled by associating time durations with process disabling and enabling operators. Espinosa *et al.*'s system is particularly geared for prototyping system performance, but is not marketed as a system for "specification".

6.4.2.7 Comparison summary

The table below summarizes the comparison of our TLOTOS with other time extensions to LOTOS. The table is based upon features 1-8 listed in section 6.4.1.

	TLOTOS	TIC	CELOTOS	T-LOTOS	Simulator supported LOTOS	Compound event LOTOS
1.access to global clock	y	n	y	y	n	n
2.events constrained in time	y	y	y	y	y	y
3.measure intra-event duration	y	n	y	n	n	n
4.expressing & composing local constraints	y	y	y	n	n	n
5.compatibility with LOTOS	y	Basic LOTOS only	y	Basic LOTOS only with action-guarded recursion	n	n
6.relative quantitative timing	y	y	y	y	y	y
7.must timing	y	y	n	y	n	n
8.ASAP, ALAP	y	n	n	ASAP only	n	n

6.5 Formal definition of TLOTOS

6.5.1 Syntax of TLOTOS

This section introduces the elements of TLOTOS syntax which are extensions of the standard LOTOS syntax.

Our concern is to integrate time extensions as unobtrusively as possible into the LOTOS syntax. The aim is to present to the TLOTOS user all the functions and sorts pertaining to time as pre-defined type definitions. These may be used as any normal type definitions. The new syntactic extensions — *time-offer*, *time-policy* and *time-establishment* — are used in a straightforward and intuitive way.

6.5.1.1 The time metric

The type definition *TimeType* in appendix C defines *TimeSort* as our metric for the measurement of time. *TimeSort* terms are isomorphic to the natural numbers.

6.5.1.2 Sets of time values

The *TimeSetType* definition defines *TimeSetSort* terms which are non-denumerable sets of *TimeSort* values. (Note that *TimeSetType* is not constructed as an *actualization* of the *Set* type, found in the standard data type library, because *Set* describes only denumerable sets.) We want the flexibility of non-denumerable sets, and their power to express infinite sets of time values. (For example, with the infinite set $\{t|t \geq 4\}$, we can specify that an event is to occur 'henceforth from time 4'.)

Sections 6.3.8 and 6.3.9 established that we need to be able to define the functions *IsIn*, *isGTAllMembersOf*, *isUpperLimited*, *Min* and *Max* over all terms of *TimeSetSort*, in order to implement *must* timing and the ASAP and ALAP time policies. In other words, we need define the functions *IsIn*, etc. for all primitive constructor functions of *TimeSetSort* terms. *TimeSetType* introduces the functions *IsIn*, etc. and defines them over the first primitive constructor function *Empty*. (The other primitive constructor functions are dealt with in *SetGeneratorFunctionsType*.)

Once *TimeType* and *TimeSetType* have been imported from TLOTOS's extended standard data type library, terms of the sorts *TimeSort* and *TimeSetSort* can be declared and used just like any normal sort.

6.5.1.3 Generator functions

The *SetGeneratorFunctionsType* definition in appendix C defines all the primitive constructors (except *Empty*) of *TimeSetSort* terms. The primitive constructors of *TimeSetSort* terms are *Empty*, *setEQ*, *setLE*, *setGE*, *setInterval* and *Union*. The constructors *setLT*, *setGT* and *Intersection* are not primitive but defined in terms of the afore mentioned primitive constructors. (In fact, it is possible to further rationalize the number of primitive constructor functions, but we have not done so for the sake of specifier convenience.)

SetGeneratorFunctionsType defines the functions *IsIn*, *isGTAllMembersOf*, *isUpperLimited*, *Min* and *Max* over all terms constructed by the primitive constructor functions.

The TLOTOS user may create more complex *TimeSetSort* generator functions. The idea is that the TLOTOS user must define complex generator functions in terms of the primitive constructor functions. This means that complex generator functions are reducible to combinations of primitive constructor functions, and hence amenable to the required application of the *IsIn*, etc. functions. (The *Intersection* function is an example of a more complex generator function which has been defined in terms of the primitive constructor functions.)

Examples later show how to use *TimeSetSort* generator functions within *time-offers*, and section 6.5.4 shows how the *IsIn*, etc. functions are used in defining the semantics of TLOTOS.

6.5.1.4 User-defined generator functions

TLOTOS users may define additional *TimeSetSort* constructor/generator functions provided that they define their functions solely in terms of the functions *Empty*, *setEQ*, *setLE*, *setGE*, *setInterval*, *Union* and *Intersection*. In this way, the *IsIn*, *isGTAllMembersOf*, *isUpperLimited*, *Min* and *Max* functions will be implicitly defined over these new user-defined *TimeSetSort* generator functions.

For example:

```
(* ************************************************************************
 * NewSetGeneratorType: contains user-defined TimeSetSort generator
 * functions
 ************************************************************************ *)
type NewSetGeneratorFunctionsType is SetGeneratorFunctionsType

   opns
      setLEofEvens:  -> TimeSetSort

   eqns
      forall t: TimeSort
         ofsort TimeSetSort
            (* We consider '0' to be even *)
            setLEofEvens(0) = SetEQ(0);
            (* We assume the existence of the function isEven *)
            IsEven(Succ(t)) =>
               setLEofEvens(Succ(t)) = setLEofEvens(t) Union setEQ(Succ(t));
            not(IsEven(Succ(t))) =>
               setLEofEvens(Succ(t)) = setLEofEvens(t);

endtype (* NewSetGeneratorFunctionsType *)
```

The user-defined function *setLEofEvens* generates a *TimeSetSort* set whose elements are even values less than or equal to a given value, of sort *TimeSort*.

6.5.1.5 TLOTOS action-denotations

We add the following *word-symbols* to [ISO89b, clause 6.1.3.1]:

asap-symbol = "**ASAP**".
alap-symbol = "**ALAP**".

The following *special-symbols* are added to clause 6.1.3.2 (and removed from *special-character* clause 6.1.2 to prevent parsing problems):

at-symbol = "**@**".
open-time-pred-symbol = "{".
close-time-pred-symbol = "}".

Alter the *action-denotation* clause 6.2.7.11 to:

action-denotation =	gate-identifier
	[experiment-offer { experiment-offer } [selection-predicate]]
	[time-offer] [time-policy] [time-establishment]
	| internal-event-symbol
	[time-offer] [time-policy] [time-establishment].
time-offer =	open-time-pred-symbol value-expression
	close-time-pred-symbol.
time-policy =	asap-symbol
	| alap-symbol.
time-establishment =	at-symbol value-identifier.

Constraint: The *value-expression* within *time-offer* must have the sort *TimeSetSort*.

Time-offers may be associated with observable and internal events. A *time-offer* is used to constrain the occurrence time of an event to one value from a given set. Such a set is generated by a *TimeSetSort* constructor function (described earlier in this subsection).

A *time-policy* may be used to specify that an event must occur ASAP or ALAP (within any time-window already established by a *time-offer*).

Time-establishment may be used to establish the occurrence time of an event, relative to the base clock. The *value-identifier*, within a *time-establishment*, is declared to be of sort *TimeSort* and is bound to the occurrence time of the associated event.

The evaluation order of terms within an *action-denotation*, is as follows:

1. Firstly, *experiment-offer* values satisfying the *selection-predicate* are negotiated and established.

2. Then, a preliminary time-window (*TimeSetSort* value) satisfying the *time-offer* is negotiated and established.

3. Finally, the *time-policy* is applied to the preliminary time window to result in the final time-window (set of possible occurrence times for the event).

Notice that this evaluation ordering allows *TimeSort* values to be negotiated as *experiment-offers* and then, within the same *action-denotation*, used within the *time-offer*. See the subsection on 'flattening *action-denotations*' for more information.

6.5.1.6 Examples of action-denotations

u {setLE(3)}

The event *u* is offered only at times which are less than or equal to 3. [20]

v ? tx: TimeSort [tx gt 5] {setEQ(2) Union setEQ(5) Union setGT(tx)}

The event at *v* is offered only when both its *selection-predicate* is satisfied and when its *time-offer* is satisfied. The *time-offer* of event at *v*, is satisfied at times 2, 5, and greater than *tx*, de-referenced. Notice that values negotiated in an

[20] For convenience we use 3 to denote *succ(succ(succ(0)))*, and similarly for other *TimeSort* terms.

event *experiment-offer* (e.g. ? *tx: TimeSort*) can be used within the *time-offer* of the same event. This is because (see sections 6.5.1.5 and 6.5.3.2) the *time-offer* is *evaluated* in an environment which includes bindings for the *value-identifiers* of the *experiment-offers* clause. Also notice that different *TimeSetSort* constructor functions (e.g. *setEQ*, *setGT*) may be combined to generate a set of *TimeSort* values.

w {setInterval(4,9)} @t1

The event *w* is offered only at times within the interval 4 to 9 inclusive. Also, if the event *w* does occur, then its actual time of occurrence will be recorded in the variable *t1*, which can then be referenced in the *action-prefix* expression following this *action-denotation*. The variable *t1* is implicitly declared to be a normal LOTOS value identifier of sort *TimeSort* (see section 6.5.3.2).

A *time-establishment* allows the quantitative time, relative to the base clock, of an event occurrence to be established. Hence this mechanism not only provides a means to *measure* duration between events (not necessarily consecutive), but also facilitates the expression of time constraints *relative* to any event — see the next example.

w {setInterval(4,9)} @t1; u {setLE(10)}; z {setEQ(t1+7)}

The event *z* is offered only at the time *t1* + 7. The previous example explained how *t1* records the occurrence time of event *w*. Therefore event *z* is offered 7 units of time after the occurrence of *w*. In this way we can state time constraints *relative* to any event — in this example, the occurrence time of event *z* is constrained relative to the occurrence time of event *w*. The intervening event *u* has been introduced to demonstrate that quantitative timing constraints can be specified between non-consecutive events (in this case, *w* and *z*).

x ASAP

The event *x* is constrained to occur ASAP (i.e. as soon as possible). Event *x* must occur at the earliest time at which the participating processes in event *x* are ready to synchronize.

In this way, we can emulate [BL91]'s ASAP semantics.

y {setInterval(4,9)} ALAP @t2

The event *y* is constrained to occur ALAP (i.e. as late as possible) within the time interval 4 to 9 inclusive. If considered in isolation, *y must* occur at time 9. However, the ALAP time may have to be earlier if this *action-denotation* is to synchronize with other *action-denotations* offering *y*. Also, the actual time of occurrence of event *y* is recorded in the variable *t2*.

6.5.1.7 TLOTOS enable-expressions

We alter the *enable-operator* [ISO89b, clause 6.2.7.6], to:

enable-operator = enable-symbol
[accept-symbol identifier-declarations [time-establishment]
in-symbol]
| enable-symbol
[accept-symbol [identifier-declarations] time-establishment
in-symbol].

Time-establishment may be used to establish the **exit** time (i.e. the δ event occurrence time) of a behaviour expression.

6.5.1.8 Examples for enable-expressions

P[h] ≫**accept** @t1 **in** Q[g](t1)

The **exit** time of behaviour expression P is recorded in the variable $t1$, which is passed as a *value-parameter* into the behaviour expression Q. $t1$ can be used within Q just like any other value term.

6.5.1.9 Preservation of relative ordering

Any event without a *time-offer* is simply constrained to occur between the immediately preceding and succeeding events. Thus TLOTOS preserves the relative ordering facility of standard LOTOS.

An event, y say, without an associated *time-offer* is interpreted as: y {setGE(0)}.

6.5.1.10 Events occurring at the 'same time'

Two or more events may occur at the same quantitative time, even if they are composed as:

x{setEQ(4)}; y{setEQ(4)}

On initial interpretation, this might seem contradictory to the use of the sequencing ';' operator. This apparent contradiction is resolved if we accept the following explanation. When two events x and y apparently occur at the same quantitative time (relative to the base clock) but are actually composed as '$x;y$', x occurs a negligible/unmeasurable duration before y, given the granularity at which we can measure time durations in the system. This explanation is satisfactory in all problems, and the expression of such negligible/unmeasurable durations between events may be useful.

6.5.1.11 Examples for parallel-expressions

For the following list of parallel expression contexts, we describe the results of negotiating time window values (*TimeSetSort* terms). (Also see the example in section 6.3.9.)

v ? tx: TimeSort [tx gt 5] {setInterval(5,tx)}; **stop** || v ! 7; **stop**

The event at v is offered only at times within the interval 5 to 7, inclusive.

Section 6.5.3.2 (flattening *action-denotations*) defines that the results of an *experiment-offer* negotiation (e.g. ? tx: TimeSort) can be used within a *time-offer* negotiation. In the example above the *experiment-offer* value identifier *tx* is negotiated to be 7. This value is then used within the *time-offer* to restrict the event at *v* to the time interval 5 to 7 inclusive.

v ? tx: TimeSort [tx gt 5] {setInterval(5,tx)}; **stop** || v ! 7 **ASAP**; **stop**

The event at *v* is offered only at the time 5 (i.e. the 'as soon as possible' time within the the interval 5 to 7).

v {setInterval(5,7)}; **stop** || v {setLE(9)} **ALAP**; **stop**

The event *v* is offered only at the time 7 (i.e. the 'as late as possible' time).

v {setInterval(5,11)} **ASAP**; **stop** || v {setLE(9)} **ALAP**; **stop**

The event *v* is offered only at times within the interval 5 to 9, inclusive. The result of any conjunction which includes both ASAP and ALAP *time-policies*, is a Normal *time-policy* (the default).

6.5.2 Formal semantics of TLOTOS

Sections 6.5.2, 6.5.3 and 6.5.4 describe the formal semantics of TLOTOS. We describe those aspects of the TLOTOS semantics which are extensions or modifications of the LOTOS as defined in [ISO89b].[21]

6.5.2.1 Overview of the definition of TLOTOS

Figure 6.15 provides an overview of the different aspects of the definition of TLOTOS. The previous subsection 6.5.1 addressed the syntax definition. The following subsections will define the static and dynamic semantics of TLOTOS.

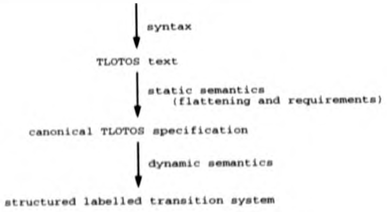

syntax
TLOTOS text
static semantics
(flattening and requirements)
canonical TLOTOS specification
dynamic semantics
structured labelled transition system

Figure 6.15: Definition aspects of TLOTOS

[21]Hence, for a complete description of the TLOTOS semantics, the reader should read this section in conjunction with [ISO89b]. However, reading this section in isolation should provide a sufficient understanding of the quantitative time related aspects of the TLOTOS semantics.

Figure 6.15 shows that a TLOTOS text, generated according to the syntax rules of section 6.5.1, is the result of the first aspect. The static semantics aspect takes a TLOTOS text and transforms it into an abstract syntax structure, known as a canonical TLOTOS specification (CTS). This transformation is carried out by a syntax directed function, called the *flattening function*. The flattening function embodies the static semantics requirements of TLOTOS; the translation of TLOTOS texts which are not in accordance with the static semantics is undefined.

A CTS consists of two related parts:

- An algebraic specification, *AS*, which contains representations of TLOTOS data types.
- A behaviour specification, *BS*, which contains representations of TLOTOS behaviour definitions.

The dynamic semantics of a CTS are defined as an interpretation of the CTS as a set of structured labelled transition systems (LTSs). Each correct substitution of the actual values for the formal parameters of a CTS is interpreted as a single LTS which serves as a model of the dynamic behaviour of the corresponding CTS instance.

6.5.3 Static semantics

The static semantics of a LOTOS specification are defined by the flattening function:

$\#.\#$: *LOTOS texts* $\rightarrow$ *canonical LOTOS specifications*

given in [ISO89b, section 7.3]. We extend this flattening function for TLOTOS, so that its signature becomes:

$\#.\#$: *TLOTOS texts* $\rightarrow$ *CTSs*

$\#.\#$ is a partial function which defines CTSs for only those TLOTOS texts that conform to the static semantic requirements.

This section provides definitions of only those aspects of the TLOTOS flattening function that significantly differ from the LOTOS flattening function found in [ISO89b, section 7.3].

6.5.3.1 Standard library

We extend the standard library *data-type-definitions* of [ISO89b, annex A] to include those *data-type-definitions* defined in section 6.5.1 and appendix C.

6.5.3.2 Flattening action-denotations

This is an extension of [ISO89b, section 7.3.4.5 clause u].

if a is an *action-denotation*,
gid is a *gate-identifier*,
$d_1, \ldots, d_n$ are *experiment-offer* occurrences,

P is a guard,
T is a *time-offer*,
h is a *time-policy*,
z is a *time-establishment*

with $a = gid\ d_1 \ldots d_n\ P\ T\ h\ z$

then $\#a\#(TE,GE,VE,scp) = \#gid\#(GE)\ d'_1 \ldots d'_n\ \#P\#(TE,VE \cup V)\ T'\ h\ z'$

where

d'_i	$= !recon(E,TE,VE,undef)$	if $d_i = !E (1 \leq i \leq n)$ where E is a *value-expression*
	$= ?e\_vid_1,\ldots,e\_vid_m$	with $\prec e\_vid_1 \ldots e\_vid_m \succ = \#id\#(TE,scp)$ if $d_i = ?id (1 \leq i \leq n)$ where id is an *identifier-declaration*

$V = \{e\_vid | e\_vid \in \#id\#(TE,scp), (d_i = ?id, 1 \leq i \leq n)\}$

Note: *Experiment-offers* are flattened as for standard LOTOS: *Value-expressions* (viz. $!E$) are *reconstructed*, and *identifier-declarations* (viz. $?id$) are flattened to extend the *value-environment*.

$T' =$	$\{recon(E,TE,VE \cup V,undef)\}$	if $T = \{E\}$ where E is a *value-expression*

Note: Flattening a *time-offer* involves *reconstructing* the *value-expression* E in the *value-environment* VE extended (by $\cup V$) with the $d_1,\ldots,d_n$ *value-identifiers*. Evaluation using this extended *value-environment* means that the TLOTOS user may reference within the *time-offer*, the variables $d_1,\ldots,d_n$ of the same *action-denotation*. E should produce a ground term of the sort TimeSetSort.

z'	$= @e\_vid$	with $\prec e\_vid \succ = \#vid : TimeSort\#(TE,scp)$ $z = @vid$ where vid is a *value-identifier*.

$V' = \{e\_vid | e\_vid \in \#vid : TimeSort\#(TE,scp)$ where $z = @vid\}$

Note: Flattening a *time-establishment* involves extending the *value-environment* with a new binding for a variable vid of sort Time-Sort.

Requirement u1: d'_i shall be defined for all i, $1 \leq i \leq n$.

Requirement u2: all vid with $e\_vid = \prec\prec vid, scp \succ, e\_sid \succ \in \#id\#(TE,scp)$ where ($d_i = ?id$ for some i $(1 \leq i \leq n)$) or ($id = vid : TimeSort$), shall be pairwise different.

6.5.3.3 Flattening enable-expressions

This is an extension of [ISO89b, section 7.3.4.5 clause e].

if beh, beh_2 are *enable-expressions*,
beh_1 is a *disable-expression*,
ifd is an *identifier-declarations*,
z is a *time-establishment*

with $beh = beh_1 \gg$ **accept** *ifd* z **in** beh_2

then $\#beh\#(TE,PE,GE,VE)$

$$= \#beh_1\#(TE,PE,GE,VE) \gg$$
$$\textbf{accept } \#\text{ifd}\#(TE, scp(beh_2))\ z' \textbf{ in}$$
$$\#beh_2\#(TE,PE,GE,VE \cup V)$$

$$func(beh,TE,PE,VE) = func(beh_2,TE,PE,VE \cup V)$$

where

$z' = @e\_vid$ with $\prec e\_vid \succ = \#vid : TimeSort\#(TE, scp)$
$z = @vid$ where *vid* is a *value-identifier*.

$V = \{e\_vid | (e\_vid \in \#ifd\#(TE, scp(beh_2)))$
or $(e\_vid \in \#vid : TimeSort\#(TE, scp(beh_2)) where (z = @vid))\}$

Note: Flattening a *time-establishment* involves extending the *value-environment* with a new binding for a variable *vid* of sort Time-Sort.

Requirement e1: $func(beh_1, TE, PE, VE) = \prec e\_vid_1.e\_sid_1, \ldots, e\_vid_n.e\_sid \succ$
if $\#ifd\#(TE, scp(beh_2)) = \prec e\_vid_1, \ldots, e_vid_n \succ$

6.5.4 Dynamic semantics

This subsection provides the definition of the semantics of a canonical TLOTOS specification $CTS = \prec AS, BS \succ$.

6.5.4.1 Structured LTS of a behaviour-expression

The structured labelled transition system $TS_{CLS}(B)$ of a *behaviour-expression* B, relative to a canonical TLOTOS specification $CTS = \prec AS, BS \succ$ is the tuple:

$$\prec S, G \cup \{i, \delta\}, AS, TT, s_0 \succ, \text{ with}$$

- S is the set of all possible states.

- $TT = \{-aTHt \rightarrow | a \in Act, T \in Q(TimeSetSort), H \in NegotiatedTimePolicySort, t \in Q(TimeSort)\}$
  with
  $-aTHt \rightarrow = \{ \prec \prec B_1, t_1 \succ, \prec B_2, t \succ \succ | D_{CTS} \vdash \prec B_1, t_1 \succ -aTHt \rightarrow \prec B_2, t \succ \}$

  where

  - $Act = \{\mathbf{i}\} \cup \{gv | g \in G \cup \{\delta\}, v \in DD^*\}$, and
    D_{CTS} is the derivation system defined in the axioms and inference rules of transition defined below.
  - TT is the set of timed transitions, i.e. TT is the set of relations $-aTHt \rightarrow$ defining the pairs of states associated with event $aTHt$.

    Note: Each transition in the set TT is attributed with event gate-name and value-negotiation information (a), occurrence-time information (t), *time-policy* negotiation information (H), and *time-offer* information (the set T). The two attributes H and T are not for *user-consumption*. These two attributes should not be included in traces, for user-consumption, produced from transition sequences. The H and T attributes are required in the definition of the axioms and inference rules, below.
  - H carries the result of negotiating a *time-policy* for an event. The table in figure 6.16 defines *time-policy* negotiation semantics.
    In this section we use the function *Negotiate(tp1,tp2)* to compute the *time-policy* which results from the synchronization of two event offers with *time-policies p1* and *p2*. The table defines this *Negotiate* function. We say that terms within the table are of sort *NegotiatedTimePolicySort*.

	Asap	Alap	Normal	Annihilated
Asap	Asap	Annihilated	Asap	Annihilated
Alap	Annihilated	Alap	Alap	Annihilated
Normal	Asap	Alap	Normal	Annihilated
Annihilated	Annihilated	Annihilated	Annihilated	Annihilated

Figure 6.16: Time-policy negotiation rules

    Notice how we use the term *Annihilated* to indicate that both ASAP and ALAP *time-policies* have been offered for the same event. If the result *Annihilated* is the outcome of a negotiation, then a Normal *time-policy* will be applied for the event (see the axioms and inference rules below).

- $s_0 = \prec B, t \succ$ is the initial state, and $t = 0$ if B is the initial process definition $pdef_0$.

The transition derivation system of a $CTS = \prec AS, BS \succ$ is the triple $D_{CTS} = \prec As, Ax, I \succ$, with

- $As = \{ \prec B, t \succ -aTHt' \rightarrow \prec B', t' \succ | B, B' \in BE, a = \mathbf{i}$ *or* $a = gv$ *with* $g \in G \cup \{\delta\}, v \in DD^*, t, T \in Q(TimeSetSort), H \in NegotiatedTimePolicySort, t' \in Q(TimeSort)\}$.

- Ax: the axioms defined in a later subsection.
- I: the inference rules defined in a later subsection.

6.5.4.2 Axioms of transition

Only a selection of the axioms are given here. The omitted cases are straightforward extensions or reductions of the following cases.

Atomic-expressions

if

B is an *atomic-expression*,
t, t' are ground terms of sort *TimeSort*

with

$\prec B, t \succ = \prec \textbf{stop}, t \succ$

then

$\prec B, t \succ \not\rightarrow$
is an axiom.

Note: No further transitions are possible from state B.

else with

$\prec B, t \succ = \prec \textbf{exit}, t \succ$

then

$\prec B, t \succ -\delta T H t \rightarrow \prec \textbf{stop}, t \succ$
is an axiom,

where

H =Asap,
$T = \{t\}$.

Note: **exit**s are forced to occur 'as soon as possible'. This ensures that any behaviour expressions which can **exit** do so immediately.

Action-prefix-expressions

if

B, B' are *action-prefix-expressions*,
t, t' are ground terms of sort *TimeSort*,
$[SP]$ is a *selection-predicate*,
g is a *gate-name*,
T is a term of sort *TimeSetSort*,
$d_1, \ldots, d_n$ are *experiment-offers*,
h is a *time-policy*,
z is a *time-establishment* instance

with

$\prec B, t \succ = \prec g d_1 \ldots d_n [SP] T h z; B', t \succ$

then

$\prec B, t \succ -g v_1 \ldots v_n T' H t' \rightarrow \prec [ry_1/y_1, \ldots, ry_n/y_n, t'/w] B', t' \succ$
is an axiom,

iff

$v_i = [t_i]$	if $d_i = !t_i (1 \leq i \leq n)$ and t_i is a ground term,
$v_i \in Q(s_i)$	if $d_i = ?x_i (1 \leq i \leq n)$ with $sort(x_i) = s_i$,
$ry_1, \ldots, ry_m$ are instances with $v_i = [ry_i]$	if $d_i = ?y_j (1 \leq i \leq n, 1 \leq j \leq m)$ and $\{y_1, \ldots, y_m\} = \{x_i \mid d_i = ?x_i, 1 \leq i \leq n\}$,

$D \vdash SP^{v}$,

Note: The requirements governing value-negotiation over *experiment-offers*, and the satisfaction of the selection-predicate, are exactly the same as those in the LOTOS standard [ISO89b].

$z = @w$ where w is a variable instance of sort TimeSort,

Note: The variable w in the *time-establishment* z, is bound to the occurrence time t' of the transition.

$T' = \{x \in TimeSort \mid (x \geq t) \wedge (x \in T)\}$,

Note: T' is the set of all possible occurrence times of the transition, given the 'present time' t, the *time-offer* T, and ignoring the *time-policy* h.

if $h = \emptyset$ then $h = Normal$,

Note: If the user has not specified a *time-policy* h for the *action-denotation*, then h assumes the default *time-policy* $Normal$.

$H = Negotiate(Normal, h)$,

Note: The negotiated *time-policy* for an *action-denotation* in isolation is just the given *time-policy* h.

if $(H = Normal) \vee (H = Annihilated)$ then $t' \in T'$
elseif $H = Asap$ then $t' = Min(T')$
elseif $(H = Alap) \wedge (isUpperLimited(T'))$ then $t' = Max(T')$
else $((H = Alap) \wedge not(isUpperLimited(T'))$ then$)$ t' is undefined,

Note: Choose the occurrence time of the transition t' out of the set of times T'. The *time-policy* H dictates how this choice is made.
If H is $Normal$ (or $Annihilated$), then choose t' to be any member of T';
elseif H is $Asap$, then choose t' to be the smallest member of T';
elseif H is $Alap$ and the set T' has an upper limit, then choose t' to be the largest member of T';
else (H will be $Alap$ and the set T' will not have an upper limit, so) t' is undefined.

else

$\prec B, t \succ \not\rightarrow$

is an axiom,

if

$t\ isGTAallMembersOfT$.

Note: No transitions are possible from state $\prec B, t \succ$ if t is greater than any member of T, i.e. it is 'too late' for any transitions to occur from this state.

where

D is the derivation system for data, generated by AS. SP' is the ground equation that is the result of the simultaneous replacement in SP of all occurrences of the variable x_i in SP which also occur contained in a $d_i = ?x_i (1 \leq i \leq n)$, by a term $r \in v_j$.

6.5.4.3 Inference rules of transition

Only a selection of the inference rules are given here. The omitted cases are straightforward extensions or reductions of the following cases.

Choice-expressions

if

B, B_2 are *choice-expressions*,
B_1 is a *guarded-expression*,
B'_1, B'_2 are *behaviour-expression* instances,
$a \in Act$,
t, t' are ground terms of sort *TimeSort*,
T is a ground terms of sort *TimeSetSort*,
H is a term of sort *NegotiatedTimePolicySort*

with

$\prec B, t \succ = \prec B_1 [] B_2, t \succ$

then

$$\frac{\prec B_1, t \succ -aTHt' \rightarrow \prec B'_1, t' \succ}{\prec B, t \succ -aTHt' \rightarrow \prec B'_1, t' \succ}$$

$$\frac{\prec B_2, t \succ -aTHt' \rightarrow \prec B'_2, t' \succ}{\prec B, t \succ -aTHt' \rightarrow \prec B'_2, t' \succ}$$

are inference rules,

Parallel-expressions

if

B, B_2 are *parallel-expressions*,
B_1 is a *choice-expression*,
B_1', B_2' are *behaviour-expression* instances,
$a, a' \in Act$,
$t, t', t_1, t_1', t_2, t_2'$ are ground terms of sort *TimeSort*,
T, T_1, T_2 are ground terms of sort *TimeSetSort*,
H, H_1, H_2 are terms of sort *NegotiatedTimePolicySort*,
$g_1, \ldots, g_n$ is a (possibly empty) list of *gate-name* instances

with

$\prec B, t \succ = \prec B_1 |[g_1, \ldots, d_n]| B_2, t \succ$

then

$$\frac{\prec B_1, t \succ -aT_1H_1t_1' \rightarrow \prec B_1', t_1' \succ, \prec B_2, t \succ -aT_2H_2t_2' \rightarrow \prec B_2', t_2' \succ}{\prec B, t \succ -aTHt' \rightarrow \prec B_1' |[g_1, \ldots, d_n]| B_2', t' \succ}$$

and $name(a) \in \{g_1, \ldots, g_n, \delta\}$.

where

$T = \{x \in T_1 \cap T_2 | x \geq t\}$,
$H = Negotiate(H_1, H_2)$,

Note: Negotiate a resultant *time-policy* by considering the two *time-policies* requested by the two behaviour expressions B_1 and B_2.

if $(H = Normal) \vee (H = Annihilated)$ then $t' \in T$
elseif $H = Asap$ then $t' = Min(T)$
elseif $(H = Alap) \wedge (isUpperLimited(T))$ then $t' = Max(T)$
else $((H = Alap) \wedge not(isUpperLimited(T))$ then$)$ t' is undefined.

Note: As expected, this inference rule ensures that a synchronization occurs only at a time agreed on by both participants.

Notice how t' is calculated from the sets T_1 and T_2, and not from considering the time values t_1' and t_2'. This is because the time values t_1' and t_2' have been pre-determined in an isolated context using H_1 and H_2 respectively. However, the occurrence times t_1' and t_2' are not necessarily valid occurrence times in this *parallel-expression* context. Hence we determine t' (the occurrence time for this *parallel-expression* context) by considering both the negotiated *time-policy* H and the sets T_1 and T_2.

exits may synchronize on δ at a time t' because our rules for *parallel-expressions* guarantee that all sub-expressions (viz. B_1 and B_2) share the same time value t. Moreover, δ occurs ASAP because of our *atomic-expression* axiom for *exit*.

$$\frac{\prec B_1, t \succ -aT_1H_1t'_1 \rightarrow \prec B'_1, t'_1 \succ, \quad (((\prec B_2, t \succ -a'T_2H_2t'_2 \rightarrow \prec B'_2, t'_2 \succ) \wedge (t'_2 \geq t'_1)) \vee (\prec B_2, t \succ \not\rightarrow))}{\prec B, t \succ -aT_1H_1t'_1 \rightarrow \prec B'_1|[g_1, \ldots, d_n]|B_2, t'_1 \succ}$$

and $\{name(a), name(a')\} \cap \{g_1, \ldots, g_n, \delta\} = \emptyset$.

$$\frac{(((\prec B_1, t \succ -a'T_1H_1t'_1 \rightarrow \prec B'_1, t'_1 \succ) \wedge (t'_1 \geq t'_2)) \vee (\prec B_1, t \succ \not\rightarrow)), \quad \prec B_2, t \succ -aT_2H_2t'_2 \rightarrow \prec B'_2, t'_2 \succ}{\prec B, t \succ -aT_2H_2t'_2 \rightarrow \prec B_1|[g_1, \ldots, d_n]|B'_2, t'_2 \succ}$$

and $\{name(a), name(a')\} \cap \{g_1, \ldots, g_n, \delta\} = \emptyset$.

are inference rules,

Note: These last two inference rules govern the independent evolution of two parts of a system. To ensure that time progresses to the same extent in both the 'active' and 'inactive' parts of the system, all parts of the system, in any particular state, share the same time value (viz. t, t', etc.).

Must timing is enforced by ensuring that any transitions occurring within the 'active' part of the system do not pre-empt the occurrence of any enabled transitions within the 'inactive' part of the system (e.g. in the last rule, $t'_1 \geq t'_2$). Also, if one part of the system has reached a state from which no transition exists (e.g. $\prec B_1, t_1 \succ \not\rightarrow$), the other part of the system may evolve alone.

Disable-expressions

if

B, B_2 are *disable-expressions*,
B_1 is a *parallel-expression*,
B'_1, B'_2 are *behaviour-expression* instances,
$a \in Act$,
$t, t', t_1, t'_1, t_2, t'_2$ are ground terms of sort *TimeSort*,
T is a ground terms of sort *TimeSetSort*,
H is a term of sort *NegotiatedTimePolicySort*,

with

$\prec B, t \succ = \prec B_1 \rhd B_2, t \succ$

then

$$\frac{\prec B_1, t \succ -aTHt' \rightarrow \prec B'_1, t' \succ}{\prec B, t \succ -aTHt' \rightarrow \prec B'_1 \rhd B_2, t' \succ}$$

and $name(a) \neq \delta$

$$\frac{\prec B_2, t \succ -aTHt' \rightarrow \prec B'_2, t' \succ}{\prec B, t \succ -aTHt' \rightarrow \prec B'_2, t' \succ}$$

$$\frac{\prec B_1, t \succ \ -\delta T H t' \rightarrow \prec B_1', t' \succ}{\prec B, t \succ \ -\delta T H t' \rightarrow \prec B_1', t' \succ}$$

are inference rules,

Enable-expressions

if

B, B_2 are *enable-expressions*,
B_1 is a *disable-expression*,
B_1' is a *behaviour-expression*,
$a \in Act$,
t, t' are ground terms of sort *TimeSort*,
T is a ground terms of sort *TimeSetSort*,
H is a term of sort *NegotiatedTimePolicySort*,
z is a *time-establishment* instance

with

$\prec B, t \succ = \prec B_1 \gg \mathbf{accept}\ z\ \mathbf{in}\ B_2, t \succ$

then

$$\frac{\prec B_1, t \succ \ -aTHt' \rightarrow \prec B_1', t' \succ}{\prec B, t \succ \ -aTHt' \rightarrow \prec B_1' \gg \mathbf{accept}\ z\ \mathbf{in}\ B_2, t' \succ}$$

and $name(a) \neq \delta$

$$\frac{\prec B_1, t \succ \ -\delta T H t' \rightarrow \prec B_1', t' \succ}{\prec B, t \succ \ -\mathbf{i} T H t' \rightarrow \prec [t'/w] B_2', t' \succ}$$

are inference rules,

Note: The variable w is bound to the **exit** time of B_1. In this way, an enabled expression can establish the time at which it becomes 'live'.

iff

$z = @w$ where w is a variable instance of sort TimeSort.

6.5.4.4 Discussion

Examples Appendix D contains example applications of the semantics defined in this section.

User-consumable output Transitions ($-aTHt\rightarrow$) in TLOTOS semantics are labelled with *time-offer* information T and *time-policy* information H. As we mentioned earlier, this information is not for *user-consumption*, but is required for the definition of the semantics. The user sees the LTS defined, or the traces produced from the transition sequences defined by the semantics. To 'tidy-up', *Time-offer* and *time-policy* information should be filtered from these, user-consumable end-products of the semantics. For convenience we have ignored this tidy-up task.

One way to filter-out *time-policy* and *time-policy* information before it 'reaches' the user is as follows. Define two types of inference schemas: schemas for internal use (I-schemas), and schemas (U-schemas) which are to be used for generating a user-consumable LTS and traces. *Time-offer* and *time-policy* information is preserved within I-schemas, but not included in U-schemas. Now to infer new axioms or inference rules we use I-schemas, because they carry all the information (including *time-offer* information T and *time-policy* information H) that we need. But we use U-schemas to define the LTS that the user 'sees', because U-schemas do not include *time-offer* and *time-policy* information. Hence, we cannot use U-schemas to infer other axioms or inference rules. Now each axiom or inference rule will infer both I-schemas and U-schemas, as the templates below illustrate.

$$\frac{\prec \mathrm{I-schema} \succ}{\prec \mathrm{I-schema} \succ}$$

I-schema used to infer more axioms and inference rules with I-schema forms.

$$\frac{\prec \mathrm{I-schema} \succ}{\prec \mathrm{U-schema} \succ}$$

I-schema is used here to infer U-schema, where U-schema is an 'end-product' for user consumption.

Set negotiation and narrowing Two features of our semantics worth an additional mention are the negotiation of *time-offer* information and the application of the *time-policy Negotiate* function. Taken together, and generalized, these two features have great potential.

The negotiation of *time-offer* information is an instance of *set negotiation* by set intersection. The application of the *time-policy Negotiate* function is an instance of what we call *set narrowing*. For each event, our semantics uses set negotiation to establish a 'preliminary' *TimeSetSort* set. Then *set narrowing* is applied (as directed by the *time-policy*) to the 'preliminary' *TimeSetSort* set to produce a 'final' *TimeSetSort* set. This 'final' set contains the possible occurrence times of the event when all *time-offers* and *time-policies* are taken into consideration.

Introducing into LOTOS generalized facilities for *set negotiation* and *set narrowing*, and making these facilities accessible to the LOTOS user, seems potentially useful. We envisage that *set negotiation* and *set narrowing* could be applied to any user defined set sorts. The user would be responsible for:

- indicating the basis for *set negotiation* (e.g. normal set union)

- defining the *set narrowing* functions (e.g. 'remove all items from the set which do not satisfy predicate P')
- defining the result of the 'synchronous application' of dissimilar *set narrowing* functions to the same set (for example, see figure 6.16 which defines the result of a 'synchronous application' of dissimilar *time-policies*).

If LOTOS was extended with *set negotiation* and *set narrowing*, one application of these facilities could be used to define at the syntax level quantitative time facilities (with ASAP and ALAP) such as we have built at a semantic level.

6.6 Mapping TLOTOS to standard LOTOS

Our previous sections have developed a time-extended version of LOTOS that we have called TLOTOS. In this section we augment this work by attempting to devise a function for mapping TLOTOS descriptions to standard LOTOS descriptions (also see [McC91b]). In essence, we are investigating if it is possible to map TLOTOS semantics *directly* to standard LOTOS semantics. Note that we could build a *TLOTOS interpreter* in LOTOS, but our intention is to to preserve the syntactic structure of a TLOTOS description in the translated LOTOS description, and vice versa.

Originally, we hoped that such a mapping function could be used to form the basis for a TLOTOS to LOTOS automatic translator tool. This tool would capture the time-related aspects of the semantics of TLOTOS source descriptions, in the syntactic structure of the derived standard LOTOS descriptions. The translation process was to be performed on the basis of a static analysis of the TLOTOS source description. This would make a translated TLOTOS description amenable to existing LOTOS tools and analysis techniques.

In this section, we investigate two possible mapping algorithms but show that neither is complete. That is to say that it is not, in general, possible to automatically and directly map (fully fledged) TLOTOS descriptions to equivalent LOTOS descriptions. However we observe that complete mapping functions do exist for restricted subsets of TLOTOS. The failure to find complete, direct mapping functions leads us to recommend that the semantics of LOTOS be extended, as defined in the previous sections. If TLOTOS semantics are not available, then we recommend the manual use of the *approaches* of the mapping functions when specifying systems with quantitative time requirements.

6.6.1 Representing time

In order to express quantitative time (as defined in sections 6.5.2 to 6.5.4) we have to associate each event with a time value. In LOTOS, we can represent *events located in time* by a number of approaches that include:

- The progression of time may be represented by the occurrence of specially designated events (*t-events*, see [AQ90]). Then the location in time of all other events can be established by considering their occurrence (ordering) relative to the t-events.

- All events could carry a *time-stamp* which denotes the location in time of their occurrence. Such a time-stamp may be part of the value structure an event (i.e. an event parameter — an ACT ONE sort in an *experiment-offer*).

These two schemes form the basis for our two mapping algorithms. Their realizations, merits and drawbacks are discussed in the following subsections.

6.6.2 Mapping algorithm using t-events

The first mapping algorithm is based on translating the *implicit*[22] quantitative time information contained in a TLOTOS description to *explicit* time information, in the form of special t-events[23]. Each t-event represents the passing of one unit of time.[24] This is most similar to the proposal in [AQ90]; similarities can also be found to work in [QAF90, vHTZ90].

The following subsubsections outline the translation to t-events, highlighting the main points of interest. We describe, for example cases, the results produced by sub-functions of our mapping function. We could describe the complete mapping function as a syntax directed function in a similar vein to the *flattening function* in [ISO89b, section 7.3]. However, describing the results of the mapping function for particular instances will suffice to demonstrate the strategy it embodies and its shortcomings, without giving an unwieldy definition of it. Also, we attempt 'corrections', re-defining parts of our mapping function — a task which is more clearly described using examples rather than an actual definitive definition.

We will use _XXXX to label a translation function which will not appear in the resultant LOTOS text.

6.6.2.1 Action-prefix expressions

The following translation of an *action-prefix* expression conveys the essence of our t-event translation strategy. Consider the translation of the following TLOTOS *action-prefix* expression:

```
TRANS_NORMAL_ACTION_PREFIX( a ?x:Nat !Succ(0) [predicate(x)] {setLE(5)} @t1;
                               B[a,b,c](x,t1), _ENV
                         )
```

Using our algorithm, this is expanded to the following LOTOS text[25]:

[22]In the sense that the supporting quantitative time mechanism is hidden to the TLOTOS user.

[23]In TLOTOS, t-events can be considered special in a similar way that δ events are in LOTOS.

[24]Of course, the relationship between real-life units of time and the interval between t-event occurrences does not have to be one-to-one, or in any other way proportional, although in general we will choose that this be so.

[25]**Note:** We have, for the sake of clarity and brevity, rationalised the translated text and omitted some context information which must be maintained by the automatic translating system.

```
    _UNIQUE_PROCESS_NAME(_ENV)[t,a,b,c](thetime,s)

where
    process _UNIQUE_PROCESS_NAME(_ENV) [t,a,b,c]
            (thetime:TimeSort,s:_XTR_SORT_ID(s ,_ENV) ) : noexit :=

          t?newtime:TimeSort [(newtime gt thetime) and
                                    not(thetime isGTAllMembersOf setLE(5))];
          _UNIQUE_PROCESS_NAME(_ENV)[t,a,b,c](newtime,s)
       []
          [thetime IsIn setLE(5)] ->
              a ?x:Nat !Succ(0) [predicate(x)];
              let t1:TimeSort = thetime in
                   _TRANS_NORMAL_ACTION_PREFIX( B[a,b,c](s,t1),
                                                      _UPDATE(_ENV) )
    endproc
```

Notice that (in the expanded text) the choice between the t-event and the *a* event is guarded so that it is impossible for the passing of time to pre empt the occurrence of event *a*. Later we shall see the shortcomings of this naive implementation of 'must' timing. However, we can see how event *a* is constrained to be offered only at appropriate times, and how the actual time of its occurrence (the value of *thetime*) is bound to the variable *t1*.

The *selection-predicate* [*newtime gt thetime*] ensures that t-events are attributed with a monotonically[26] increasing quantitative time.

If we look at the flattening function for TLOTOS *action-denotations*, given in section 6.5.3.2, we can see that the *time-offer* should be evaluated in an environment which is already enriched with *value-identifiers* from the *experiment-offers*. However, this is not faithfully replicated by the above translation function. This is because of the order in which the translation function needs to use information from the *time-offer* and *experiment-offers*. The translation function needs to use information from the *time-offer* to define the *selection-predicate* constraining the t-event. This information must be available before the *experiment-offers* of the *a* event are negotiated. Therefore, the mapping function (unlike the semantics in section 6.5.3.2) cannot support the evaluation of the *time-offer* in of the light of *experiment-offer* negotiation. To try to do so would lead to a *catch-22* situation. This does not arise in the semantic definitions in section 6.5.3.2, because a single transition conveys both time information and value negotiation information, whereas our translation strategy uses two transitions to convey the same amount of information.

Internal (**i**) events are treated similarly to other TLOTOS events; t-events are no substitute for **i** events — they are conceptually different. The former represents the passing of time, the latter represents some spontaneous transition within the system. Therefore, the use of the **i** event to represent the passage of time has the disadvantage that time-related properties cannot be proved if the internal event is used for other purposes also. For this reason, the mapping function parameterizes the resultant LOTOS specification with the t-event gate (guarding against the relabelling of t-events as **i** events).

[26] When a t-event *t!n* occurs, all subsequent t-event occurrences will be of the form *t!m*, where $m > n$.

Already, we have encountered problems with this mapping strategy. Nevertheless we continue with our investigation, hoping either that these problems can be later resolved, or that insight into the semantics of TLOTOS be reward enough from this investigation.

6.6.2.2 Parallel expressions

All parallel behaviour expressions must synchronize on t-events in order to share a common time.[27] This implies that:

```
_TRANS_PARALLEL( choice-exp |[gate-id-list]| parallel-exp, _ENV )
```

will be translated to:

```
    _TRANS_CHOICE( choice-exp, _ENV )
|[t,gate-id-list]|
    _TRANS_PARALLEL( parallel-exp, _ENV )
```

The *_TRANS_NORMAL_ACTION_PREFIX* described above realised 'must' timing for an *action-prefix* expression considered in isolation. However, this naive approach did not consider the consequences of placing the result of a *_TRANS_NORMAL_ACTION_PREFIX* in parallel with other behaviour expressions. If the translation example shown in the previous subsubsection was placed in the context of parallel behaviour, "synchronism deadlock" [AQ90] may occur if neither of the *selection-predicates* can be satisfied for event *a* and the t-event. Then, because of the synchronous basis of our translation strategy, this would block the progression of time throughout all other expressions combined in parallel with our example *action-prefix* expression. The desired effect from 'must' timing is that time is allowed to progress while it does not pre-empt an otherwise possible event. This means that the *_TRANS_NORMAL_ACTION_PREFIX* ought not to block the progression of time if event *a* is not a possible event. This is not the case.

Unfortunately, there is no solution to this problem within our t-event mapping strategy. The crux of the solution would involve establishing if an event is enabled (i.e. 'possible') and to block the progression of time (to force it to occur) only if it is enabled. A mapping function implementing this solution, would produce LOTOS text which bore little resemblance to the original TLOTOS text. In obedience to the aim stated at the start of this section – to preserve the syntactic structure of a TLOTOS description in the translated LOTOS description, not to build a *TLOTOS interpreter* in LOTOS – we do not pursue this solution. Instead, we drop our attempt to enforce 'must' timing in favour of avoiding synchronizm deadlock. To do this we alter the *_TRANS_NORMAL_ACTION_PREFIX* function to have it not generate the *isGTAllMembersOf* predicate, so that the progression of time may proceed independently of any other concerns.

[27] TLOTOS is based on a "synchronous model" of time.

Synchronous exits Translating TLOTOS terminating (**exiting**) behaviour expressions combined by a parallel operator poses yet another problem. Thus far we have seen that in the LOTOS text (translated from TLOTOS) we explicitly pass each behaviour expression the current time value (via the *thetime* parameter). To establish this current time we need to be able to determine the exiting time of the enabling behaviour expression. If the enabling expression is a set of synchronously exiting parallel behaviour expressions we must devise some means of negotiating the final synchronous exit time of the complete parallel expression.

In an attempt to solve this problem we replace all such synchronizing exits with special *WAIT* processes of the form:

```
process WAIT[t](thetime:TimeSort) exit :=
        exit(thetime)
    [] t?newtime:TimeSort; WAIT[t](newtime)
endproc (* WAIT *)
```

The *WAIT* process offers to exit immediately with the time of the last event in the instantiating behaviour expression. However if exit synchronization at this time with the other parallel expressions is not possible, the *WAIT* process offers to synchronize on t-events to update its 'exit time' and recurses, continually trying to synchronize its **exit** with the other parallel expressions. Thus in this way, any one behaviour expression, in a set combined by a parallel operator, can wait until all other behaviour expressions in the set are ready to terminate with it.

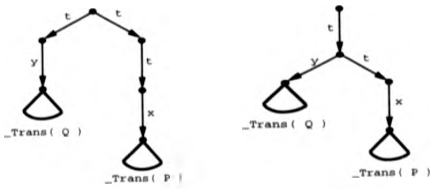
Of course, in general the *WAIT* process must be tailored to the exact functionality of its context, i.e. the *WAIT* process must offer the same list of **exit** values.

WAIT processes represent yet another departure from the TLOTOS semantics, section 6.5.4.2. In section 6.5.4.2's semantics, an **exit** is forced to occur ASAP, but this ASAP urgency is not enforced by using *WAIT* processes.

6.6.2.3 Choice expressions

The semantic definition for *choice* expressions, in section 6.5.4.3, describes a *deferred choice* model. What we mean by *deferred choice*, versus *immediate choice*, can be seen by considering the following simple example.

How should: $(x\{setLE(2)\}; P)[](y\{setLE(1)\}; Q)$ be translated? Two possibilities for time extended semantics are immediate or deferred choice. For *immediate choice* the translation of the above expression would mimic behaviour tree (a) in figure 6.17.



(a) Immediate Choice (b) Deferred Choice

Figure 6.17: Choice behaviour trees

In behaviour tree (a) we can see that two t-events are offered[28] at the choice statement. When one of these occurs it immediately determines which of the events x or y can subsequently occur. One interpretation of these semantics is that since we often choose to have time-constrained events mark the finish of actions with durations, the description is saying that the actions corresponding to the events x and y mutually exclude one another. Immediately one of these actions starts to happen the other action (and hence the other representing event) cannot occur.

Translating the same time-extended choice statement as above, but using the *deferred choice* strategy (as in the semantic definition of section 6.5.4.3) will result in the behaviour tree (b) shown in figure 6.17. This defers as late as possible the decision as to whether or not an event such as x or y will occur. Events x and y still mutually exclude one another but, in this instance, if event y does not occur at time 1, event x may still occur at time 2. In the immediate choice translation, if event y was offered but did not occur at time 1, event x would never be offered. (Compare the behaviour trees in figure 6.17.)

Of these two time-extended semantic models for choice, deferred choice seems the closest to the standard **LOTOS** choice semantics, and also is the closest to the *intuitive* interpretation of the **TLOTOS** syntax for *choice* expressions. Deferred choice is the model that we have adopted for **TLOTOS** in section 6.5.4.3.

Translating **TLOTOS** deferred choice expressions is not as straightforward as it may initially seem. The algorithm must identify all ***guarded-expressions*** which form direct alternatives to each other. Each set of such alternative ***guarded-expressions*** we call a 'choice set'. For example, the ***guarded-expressions*** whose initial events are x, y and z in the following **TLOTOS** fragment form a choice set.

```
w; ( x{setLE(2)}; Q [] y{setLE(4)}; R [] S[z] )
where
   process S[z]  noexit := z{setLE(3)}; P endproc
```

[28]The t-event may cause non-determinism if composed in parallel with itself, as any other events may. We might interpret a choice between two t-events as a choice between different 'time streams'.

Translating the choice set:

```
_TRANS_CHOICE_SET( x{setLE(2)}; Q [] y{setLE(4)}; R [] S[z], _ENV )
```

gives us:

```
_UNIQUE_PROCESS_NAME(_ENV)[t,x,y,z](thetime)

where
   process _UNIQUE_PROCESS_NAME(_ENV)[t,x,y,z](thetime:TimeSort) : noexit :=
            t?newtime:TimeSort [newtime gt thetime];
              _UNIQUE_PROCESS_NAME(_ENV)[t,x,y,z](newtime)
       [] _TRANS_CHOICE_ALT_ACTION_PREFIX( x{setLE(2)}; Q, _ENV )
       [] _TRANS_CHOICE_ALT_ACTION_PREFIX( y{setLE(4)}; R, _ENV )
       [] _TRANS_CHOICE_PROC_INSTANT( S[z], _ENV )
   endproc
```

If an *action-prefix* element in a choice set is not referenced as a direct choice alternative, we translate it using *_TRANS_NORMAL_ACTION_PREFIX* as described previously. However, where such an *action-prefix* is referenced as a choice alternative, we translate it as shown below.

```
_TRANS_CHOICE_ALT_ACTION_PREFIX( y{setLE(4)}; R, _ENV )
```

This expands to:

```
    _UNIQUE_PROCESS_NAME(_ENV)[t,y](thetime)
    ...
where
    process _UNIQUE_PROCESS_NAME(_ENV)[t,y](thetime:TimeSort) : noexit :=
        [thetime IsIn setLE(4)] ->
                    y;
                    _TRANS_NORMAL_ACTION_PREFIX( R, _ENV )
    endproc
```

Notice how the process generated by *_TRANS_CHOICE_SET* implements deferred choice by always offering the t-event, and offering the other events when appropriate. The process generated by *_TRANS_CHOICE_ALT_ACTION_PREFIX* significantly differs from the process generated by *_TRANS_NORMAL_ACTION_PREFIX* in that it does not itself offer t-events as an alternative to its *y* events, but instead relies on the process generated by *_TRANS_CHOICE_SET* for this t-event alternative. If the process generated by *_TRANS_NORMAL_ACTION_PREFIX* were to offer an alternative t-event and this occurred, all the other alternative events in the process generated by *_TRANS_CHOICE_SET* would then be excluded from ever occurring. This would not reflect our intended semantics for the deferred TLOTOS *choice* expression.

6.6.2.4 Disable expressions

Consider the translation of a TLOTOS disable expression:

```
_TRANS_DISABLE( (b{setLE(2)}; B[b,c]) ▷ D[d], _ENV )
```

This is translated to:

```
   _UNIQUE_PROCESS_NAME(_ENV)[t,b,c,d](thetime)

where
   process _UNIQUE_PROCESS_NAME(_ENV)[t,b,c,d](thetime:TimeSort) : noexit :=
          t?newtime:TimeSort; _UNIQUE_PROCESS_NAME(_ENV)[t,b,c,d](newtime)
      [] [thetime IsIn setLE(2)] -> b; _TRANS_DISABLE( B[b,c] ▷ D[d], _ENV )
      [] _TRANS_DISABLING_PROC_INST( D[d], _ENV )
   endproc
```

Notice how *_TRANS_DISABLE* allows disabling at any instant, by offering the disabling expression as an alternative to all events.

_TRANS_DISABLING_PROC_INST translates all the possible events in the disabling expression much in the same way as *_TRANS_CHOICE_ALT_ACTION_PREFIX* does, thus relegating the responsibility of updating time (via t-event synchronization) to the disablable expression.

Remember that our aim is to be able to take any TLOTOS description, statically analyse it and translate it into an equivalent finite LOTOS description. Unfortunately it is not, in general, possible to translate TLOTOS *disable* expressions which are embedded inside a recursive definition[29] to equivalent finite LOTOS. Two possible means of overcoming this difficulty are:

- Restrict the expressive power of TLOTOS by either forbidding the expression of disable expressions inside recursion, or by altering the disable operator to force the TLOTOS user to explicitly state at what times disabling may occur, e.g. [*time_constraint*>. (The times at which disabling may occur are immediately derivable from [*time_constraint*>. This is in contrast to the ▷ $P[x]$ form of the disable operator, for which we would have to analyse $P[x]$ to establish such times.)
- Integrate the translation algorithm into LOTOS simulation/expansion tools [QPF89, Joh89]. In effect this means that we rewrite the expansion theorems [ISO89b, section B.2.2] so that expansion of TLOTOS expressions yields choice LOTOS (with data values) and the appropriate placing of t events.

6.6.2.5 Establishing time-policy information

Sections 6.3.3, 6.3.8, 6.3.10 and 6.5.4.3 describe how TLOTOS supports the notions of 'must' time and time-policies. Once an event becomes 'enabled' (i.e. all participants

[29] For example, expressions such as.

$P[x,y]$ where process $P[x,y]$: noexit := $x\{setLE(2)\}$; $P[x,y]$ ▷ $D[y]$ endproc in the event are prepared to synchronize and negotiate *experiment-offer* values which satisfy the conjunction of *selection-predicates*), 'must' timing ensures that the event will be fired (given that it is not an alternative in a choice set), and the negotiated time-policy will dictate the time value, within its time window, at which the event will occur (e.g. ASAP, ALAP or Normal). Supporting 'must' timing and time-policies is not possible within the framework of our t-event mapping algorithm.

One obvious approach towards supporting 'must' timing and time-policies is as follows. The mapping functions could introduce a LOTOS *Arbitration* process for each TLOTOS gate. Each TLOTOS event, a say, could be realised as a set of LOTOS a_{offer} events, a single $a_{collected}$ event and a single a_{fired} event. Each a_{offer} event would convey *experiment-offer*, *selection-predicate*, *time-offer* and *time-policy* information pertaining to a single a event offer. The occurrence of the $a_{collected}$ event would indicate that all the a_{offer} events have occurred. The LOTOS a_{fired} event would occur at (what would have been) the chosen firing time of the original TLOTOS a event, and would carry (what would have been) the negotiated *experiment-offer* values of the original TLOTOS a event. The *Arbitration* process, for (what would have been) a single occurrence of a TLOTOS a event, would synchronize with each a_{offer} event in turn, to collect all *experiment-offer*, *time-offer*, etc. information pertaining to the TLOTOS a event. Then $a_{collected}$ event would occur to indicate that all this information had been collected. From this collected information, *Arbitration* would offer an a_{fired} event at the appropriate time in respect of the *time-offer* and *time-policy* information, and with the appropriate event parameter values in respect of *experiment-offer* and *selection-predicate* information.

To help clarify the approach just outlined, appendix E sketches an example translation using this approach. Under the approach, the TLOTOS fragment in appendix E.1 would be (approximately) translated to the LOTOS fragment in appendix E.2. Notice how the $a_{collected}$ fulfills its purpose because it cannot synchronize until all a_{offer} events have occurred. While there are a_{offer} events still to be collected, *Arbitration* permits time to progress unconstrained. However, once $a_{collected}$ occurs, and *Arbitration* computes that the a event is firable, it prevents time from progressing beyond the firing time of the a event, thus enforcing 'must' timing. Also, it seems that the *Arbitration* process can support time-policies. It can negotiate a 'preliminary' time window from the collected *time-offer* data, for the occurrence of event a. Then, from the collected *time-policies*, the resultant *time-policy* can be computed and applied to the 'preliminary' time window in order to obtain the actual time window in which event a is to occur.

However, this solution and variations on it are fraught with difficulties. For example, unless we implement another mechanism which prioritizes a_{offer} and $a_{collected}$ events over t-events, we cannot ensure that a_{offer} and $a_{collected}$ events will occur as soon as possible. This means that by the time the *Arbitration* process receives the information conveyed by these events, it may be too late to offer the corresponding a_{fired} event.

Another problem is the representation of each TLOTOS event offer as three complementary LOTOS events. This scheme creates sequencing problems. For example, we cannot naively translate: $a; P[]b; Q$ as: $a_{offers}; a_{collected}; a_{fired}; P[]b_{offer}; b_{collected}; b_{fired}; Q$. The a_{offer} and b_{offer} events would mutually exclude one another, when in fact we want the a_{fired} and b_{fired} events to mutually exclude one another. Solutions to this particular problem lead to problems elsewhere. We could solve these, but for little gain. The result from solving all these problems would a mapping function which described most of the TLOTOS semantics in LOTOS syntax. And we hoped for a much more direct mapping between TLOTOS and LOTOS than this. The next subsection briefly examines the use of time stamps, the alternative to using t-events as the basis for a mapping algorithm.

6.6.3 Mapping algorithm using time-stamps

This mapping algorithm translates the *implicit*[30] quantitative time information in TLOTOS descriptions to *explicit* time information (time-stamps) incorporated into the value structure of events. Thus TLOTOS event offers such as:

```
a ? x:X [x eq y] {setInterval(3,t1)};...
b ! z {setLE(10)} @t2; ..
a;...
```

are mapped (approximately) to the LOTOS text:

```
a ? t : TimeSort ? x:X [(x eq y) and (t IsIn setInterval(3,t1))]; .
b ? t2 : TimeSort ! z [t IsIn setLE(10)]; .
a;...
```

To impose a proper quantitative time ordering on these events, a global time process must also be composed in conjunction with the rest of the translated system description. This global time process continually offers to synchronize on *all* events in the system, negotiating a monotonically increasing[31] quantitative time-stamp for these events.

However, LOTOS allows the dynamic declaration of new gates (using **hide**), which makes it impossible to pre-determine the set of all possible system events from simple static analysis of the TLOTOS text. It is also generally impossible to 'dynamically evolve' such a global time process, i.e. to establish an initial global time process which synchronizes on observable gates, and to then reconfigure this time process on-the-fly as each **hide** operator in the given TLOTOS system is realized. This is because it proves impossible to manage the synchronization between an initial global time process and its newly evolved gates.

It is possible, through static analysis of any TLOTOS description, to pre-determine a set of observable *action-denotations* (*gate-identifiers* together with *experiment-offer* structures) such that this set has the potential to synchronize with any observable events in the system. This set can then be used to construct a global time process in conjunction with the rest of the processes in the translated system.[32] A global time

[30]In the sense that the supporting quantitative time mechanism is hidden to the TLOTOS user

[31]In the sense that when an event time-stamped t occurs, all subsequent event occurrences will be time-stamped $t+n$, where $n \geq 0$

[32]Establishing whether *action-denotations* (apart from explicit I *action-denotations*) can never be *realized* as observable events is, in general, undecidable. This means that a translation algorithm would produce a global time process which contains superfluous synchronisation offers. Although not elegant, this in itself does not affect the correctness of the resulting specification.

process for the example above is:

```
process GLOBAL_TIME[a,b](thetime:TimeSort)  noexit :=
        a!thetime?v:X; GLOBAL_TIME[a,b](thetime)
    [] b!thetime?v:Z; GLOBAL_TIME[a,b](thetime)
    [] a; GLOBAL_TIME[a,b](thetime)
    [] GLOBAL_TIME[a,b](Succ(thetime))
endproc
```

For simulation an internal event would have to be introduced to guard the recursion, but the principle is that if an event happens at time 3 (say) then the next event may happen at time 5 (say) without the specification having to explicitly *move* in constant duration steps through the time series 3 to 5, in this case.

6.6.3.1 Time constraints on observable events

We have just seen that this time-stamp based mapping algorithm can deal only with TLOTOS descriptions in which quantitative time relations are expressed *only* over observable events. Therefore, similarly to the t-event based algorithm, our development of the stamp-stamp based algorithm has already run into difficulties (and we have not yet considered supporting TLOTOS features such as 'must' timing and time-policies).

However, before abandoning the time-stamp based algorithm, we make a few interesting observations on the effects of limiting the expression of quantitative time relations over only observable events.

Most authors advocate the constraint-oriented style for high level specifications in view of its assertional characteristics. The constraint-oriented style concentrates on observable events. We believe that limiting the expression of quantitative time relations to *only* observable events is not always sufficient for the development of specifications, but its discipline does provide lessons in the development of 'good' specifications.

Consider writing a specification for a system which, after the user presses a button, will either two seconds later turn on a green light or three seconds later turn on a red light. With a restricted TLOTOS, where we could express quantitative time relations over only observable events, we would write:

```
press_button @t1;
(
    i; greenlight{setEQ(t1+2)}; exit
[]
    i; redlight{setEQ(t1+3)}; exit
)
```

Without this restriction we might have considered writing:

```
press_button @t1;
(
    i {setEQ(t1+2)}; greenlight; exit
[]
```

```
    i {setEQ(t1+3)}; redlight; exit
)
```

In this particular example, the discipline of restriction is welcome, since the second 'solution' is not a correct reflection of the requirements. The second solution specifies at what time the system determines (invisibly) which one of the two possible behaviour paths to take after the *press_button* event. Moreover, the second solution does not actually specify that the *greenlight* and *redlight* events are to occur two and three seconds respectively after the *press_button* event. This example emphasises the point that for many "specifications" it is sufficient to state relations (including quantitative time relations) among only observable events.

6.6.4 Conclusions from this section

The general objective of our work in this section was to gain a better understanding of the relationship between TLOTOS and LOTOS. Our more specific objective was to devise a general algorithm for mapping TLOTOS to LOTOS. We explored two possible algorithms — one based on *t-events* and the other on *time-stamps*.

Neither algorithm has been found to be complete in the sense that neither is powerful enough to map complete TLOTOS (as defined in sections 6.5.1 to and 6.5.4) descriptions to semantically equivalent finite LOTOS descriptions. Nevertheless, this work has been useful because the algorithms provide a basis for manually describing quantitative timing constraints in LOTOS. We believe that the *approaches* which the algorithms embody are useful for the specification of timing aspects of distributed systems.

This work has exposed many interesting problems, such as how to implement 'must' timing, time-policies and priority using Standard LOTOS. This work has been useful as a comparison between the expressive power of LOTOS and TLOTOS, and has led us to appreciate the need to enhance LOTOS (to TLOTOS) at the semantic level (rather than via a syntactic mapping algorithm).

6.7 Timing aspects of the CIM-OSA IIS revisited

In section 6.2 we used examples of the CIM-OSA IIS X_Service and X_Service_Agent to demonstrate the inadequacy of LOTOS for capturing quantitative timing concerns. To correct this inadequacy, sections 6.3 to 6.6 have concentrated on enhancing standard LOTOS to TLOTOS (LOTOS with quantitative time facilities). Now we return to our original CIM-OSA IIS examples, and provide complete and formal descriptions of their timing aspects using TLOTOS.

6.7.1 TLOTOS description of the X_Service

The TLOTOS specification, Xsrv1T in appendix F.1 is a direct reflection of the informal requirements for the X_Service, given in section 6.2.

6.7.2 TLOTOS description of the X_Service_Agent

The TLOTOS specification, Xage1T in appendix F.3 is a direct reflection of the informal requirements for an X_Service_Agent, given in section 6.2.

6.7.3 TLOTOS description of the extended X_Service_Agent

To provide further evidence of the expressive power of TLOTOS, we present the following TLOTOS description of the X_Service_Agent embellished with X_Management functionality. This specification is an abstraction of the specification S_4 discussed in section 5.5.

The X_Management functionality requires that:

- All X_Service_Agents respond appropriately to a *Closedown* broadcast message from the X_Service_Agent_Manager.
- All X_Service_Agents must terminate simultaneously on the *Closedown* event.
- The *Closedown* event must occur within the time period specified by the X_Service_Agent_Manager.
- If an X_Service_Agent is processing an X_ACCP *Request* when the *Closedown* broadcast arrives, the X_Service_Agent must wait until as late a time as possible before succumbing to the *Closedown* request. (The X_Service_Agent must, still, *Closedown* simultaneously with the other Agents.) This is to give the current X_ACCP *Request* the best chance of completing.
- If the X_Service_Agent is not processing a X_ACCP *Request* when *Closedown* broadcast arrives, the X_Service_Agent merely complies in executing the *Closedown* request within the time interval specified by the Manager, and at the same time as all the other X_Service_Agents.

The TLOTOS specification, Xage2T in appendix F.4 is a direct reflection of the informal requirements for the extended X_Service_Agent given in the paragraph above.

Considering specification Xage2T, notice how easily TLOTOS allows us to describe the actual *Closedown* behaviour. The above requirements suggest a complex negotiation mechanism for broadcasting the *Closedown* message, establishing a *Closedown* time, and finally executing the *Closedown* event. A mechanism for supporting this negotiation of an event time is already a feature of the TLOTOS semantics, hence the simple syntactic description.

Although the TLOTOS text for the *Closedown* itself is very simple, there are three instances of this text. The reason for this lies in the requirement for the Agent to *Closedown* ALAP while processing an X_ACCP *Request*, or *Closedown* at any time (compliant with the Manager) if not. Occurrence of either the second or third (in textual order) of these instances represents the case of *Closedown* while the Agent is not processing an X_ACCP *Request*. An occurrence of the first instance represents the case where the Agent is processing a X_ACCP *Request*.

The ALAP *time-policy* ensures that the X_Service_Agent delays to as late as possible the *Closedown* event if that Agent is currently processing an X_ACCP *Request*. In the X_Service_Agent_Manager process, the *Closedown* event will be associated with an appropriate *time-offer* to ensure that the *Closedown* event *must* occur within a certain time period (this is not illustrated). Looking back to specification S_8 in section 5.5, we see that all Agents, together with the Manager, synchronize on the X_Mgnt gate, thus ensuring that all the Agents *Closedown* simultaneously.

6.7.4 Discussion

Close scrutiny of these examples reveals questionable behaviour. For example:

- The first i event in the X_Service specification Xsrv1T has an attached ASAP *time-policy*. Since this event is not an interaction, we know that if this i event occurs, the ASAP *time-policy* will always force it to occur at the same quantitative time as the immediately preceding X_ACCP!Req event occurrence. Therefore, this X_Service specification will never display behaviour where the X_ACCP!Res!data2 event is offered at a time $t2$ where $(t2 > t1) \wedge (t2 \in \mathit{setLE}(t1 + \mathit{timeout\_period}))$. Examining the informal requirements (section 6.2, point 3), we would expect this to be the legitimate behaviour. However, point 3 also asks that the X_Service compute and offer an X_ACCP!Res!data2 event ASAP — the conjunction of this requirement, and the requirement that X_ACCP!Res!data2 *must* start being offered within the period $t1 \ldots t1 + \mathit{timeout\_period}$, is the reason for this dilemma. We have tried to reflect both of these requirements in the specification of the X_Service.

  One possible resolution of this dilemma is to assume that the requirement 'the X_Service compute and offer an X_ACCP!Res!data2 event ASAP' can only really be reflected in the X_Service_Agent specification. After all, this requirement deals with computation, which is an Agent issue and not an issue for the X_Service *interface-definition*. Thus, our conclusion is that the informal requirements are ill-stated: the computation urgency requirement should be placed in the requirements for the X_Service_Agent, and not in the requirements for the X_Service. To reflect this conclusion, the TLOTOS specification Xsrv2T in appendix F.2 has no ASAP urgency associated with the first i event.

  General Point: Changing the informal requirements to reflect the findings of a less abstract description suggests a symbiotic dependency between these two descriptions. This symbiotic dependency between the informal requirements and a less abstract description (and, more generally, between any two descriptions at different abstraction levels) is not a surprise. Abstract models of development life-cycles predicate iterative loops in development, and cognitive studies [Vis90] show that the developer continually jumps between different abstraction levels, changing descriptions at one level to reflect findings at another (higher or lower) level. This is an *experimental* approach to design where validation feedback may lead to modifications to both the "specification" and the "implementation" [Bri91].

- In the X_Service_Agent_Ext specification Xage2T, the first instance of *Closedown* text is composed with a *disable-operator*. This means that the ALAP *Closedown* event could occur immediately after the X_ACCP!Res!data2 (an event which indicates that the Agent has finished processing an X_ACCP *Request*). The question is, does this violate the requirements, which state that the *Closedown* event should occur ALAP only while the Agent is currently processing an X_ACCP *Request*?

  The specification Xage3T in appendix F.5 is a re-structured X_Service_Agent_Ext specification. In this alternative specification, four instances of the *CloseDown* text are used, to the effect that an ALAP *CloseDown* event can no longer immediately follow an X_ACCP!Res!data2 event. Is this alternative solution (Xage3T) a more precise reflection of the requirements than solution Xage2T? Or, to put the question another way, can we verify or test that one of these alternatives is a more precise reflection of the requirements than the other? To answer this question we need to develop a theory of verification or testing for TLOTOS.

  For answers to this and similar questions we turn to the testing theory for TLOTOS introduced in the next section.

6.8 Testing relations for TLOTOS

Appendix G forms an annex to this chapter, to take the work on TLOTOS a stage further by proposing and examining useful TLOTOS testing relations. We define TLOTOS testing relations as extensions of Standard LOTOS testing relations. We take relations such as testing congruence and equivalence, **cred**, **cext** and **red**, demonstrate their application for a few small but interesting examples, and show that these TLOTOS relations yield sensible and intuitive results. Then we use these relations to test that our CIM-OSA example specifications, appendix F, are satisfactorily related.

6.9 Summary

The primary objective of this chapter was to develop an extended version of Standard LOTOS for the formal specification of quantitative timing concerns in distributed systems.

The chapter began by showing the inadequacies of Standard LOTOS for the specification of timing requirements. Then we investigated, using a derivative of arc-timed Petri Nets, the language facilities needed for the specification of timing requirements. A set of quantitative time features were distilled from the findings of this investigation, and a proposal was made to incorporate these into an extended version of LOTOS we called TLOTOS. We contrasted our TLOTOS with other existing proposals in this area and found that TLOTOS compared favourably.

The syntax and semantics of TLOTOS were defined as extensions of the LOTOS syntax and semantics. TLOTOS semantics define a global, discrete clock which can be used both to force (using 'must' timing) events to occur at specific times, and to measure the intervals between event occurrences. TLOTOS introduces *time-policies*, i.e. ASAP ('as soon as possible' corresponding to "maximal progress semantics") and ALAP ('as late as possible).

Another facet of this work was an attempt to devise an algorithm for mapping TLOTOS to Standard LOTOS. No satisfactory algorithm for automatically mapping TLOTOS to LOTOS could be found. Nevertheless, this work proved useful because the algorithms tried provide a basis for manually describing quantitative timing concerns in Standard LOTOS. It also provided a comparison between the expressive power of LOTOS and TLOTOS.

The chapter concluded with examples demonstrating the power of TLOTOS for the capture of quantitative timing constraints. Also, the chapter refers to an annex (appendix G) which shows how TLOTOS specifications can be tested under extended definitions of the LOTOS testing relations, to yield sensible and intuitive results.

Chapter 7

Formal specification of probability for distributed systems

This chapter extends LOTOS with features for the specification of probabilistic aspects of distributed systems. We begin by defining extensions to the LOTOS syntax and semantics to produce a probabilistic version of LOTOS, we call PbLOTOS. PbLOTOS has a probabilistic choice operator for specifying probability distributions over a set of internal probability transitions.

The definition of LTSs (Labelled Transition Systems), is extended to define NP-LTSs (LTSs which may contain both non-deterministic and probabilistic transitions) and P-LTSs (LTSs which contain only probabilistic transitions). We use NP-LTSs as a semantic model for PbLOTOS.

We consider that a PbLOTOS specification (an NP-LTS) describes a set of probabilistic implementations (P-LTSs). Then, upon this basis, we define an implementation relation (a pre-order), called *probabilization*, for PbLOTOS. We show how the probabilization relation and variants of it can be used as conformance relations in the development of probabilistic systems. We conclude by laying the foundations of a statistical testing framework for establishing whether a probabilistic implementation (a P-LTS) is a valid implementation of a PbLOTOS specification (an NP-LTS), according to the probabilization relation.

7.1 Introduction

Process algebras provide a useful framework for reasoning about concurrent and nondeterministic behaviours. Recently, the reasoning power of process algebra has been extended to cover probabilistic behaviour [BIM88, BM89, LS89, vGSST90, GJS90, HJ89, HJ90, Han90]. The probabilistic aspects of distributed systems are often as important as, or inseparable from, the so-called functional aspects. In this chapter we extend the LOTOS language and theory to facilitate the specification of probability information.

In this thesis our only concern is to construct an *abstract model of probability* which is useful in the description of distributed computing systems. We are not concerned with "philosophical" problems asking "what probability really is", or "how probability should be represented", etc. We defend the abstract model of probability, developed in this section, on the basis that it can support applications such as the specification of reliability, the description of expected results from statistical tests, performance metrics for the specification of averages and limits, etc. Our model (language constructs and supporting theory) is based on existing work on process algebras and probability (e.g. [LS89, HJ89, HJ90, BM89]). Also, our model of probability is realised by a small number of extensions to LOTOS syntax and semantics. These extensions are intuitive, and do not conflict with the observational, (de)compositional reasoning supported by LOTOS.

7.1.0.1 (De)compositional reasoning

Usually we construct complex systems from simple, possibly pre-defined components (or constraints). Often we know about the reliability or statistical behaviour of such components. Given this information about individual components, we would like to be able to infer the probabilistic behaviour of the system as a whole. This is one aspect of the (de)compositional reasoning property that we would like our model of probability to support.

7.1.0.2 Distributed negotiation of probability

Another important aspect of (de)compositional reasoning for probabilistic systems is the question of how a probability distribution over a set of events is negotiated among the processes which synchronize on events in this set. Solutions to this problem, of distributed negotiation of probability distributions, usually involve the idea of a "normalization function" [vGSST90, GJS90]. The normalization function is a global agent which arbitrates probability distributions such that the composite behaviour of the system remains *stochastic* [GJS90]. This problem is not an issue for the model of probability presented in this thesis. In our model, probabilistic decisions are represented by special internal transitions which carry probability values. Because these transitions are internal, the probability values cannot be negotiated on-the-fly and so the problems of distributed negotiation of probability distributions is not applicable. Adopting this model, we sacrifice some expressive conciseness, but lose no absolute expressiveness (as shown in [vGSST90]). Also, we feel that the idea of having a particular, global "normalization function" is an arguable concept.

7.1.0.3 Non-deterministic and probabilistic systems

Section 4.4.2.6 described the important rôle played by non-determinism in the specification of systems. However, many of the existing proposals (e.g. [GJS90, vGSST90]) extend process calculi with probability information, at the expense of non-determinism. They replace language expressions for non-deterministic transitions by expressions for probabilistic transitions. We introduce probability features into LOTOS without sacrificing the non-deterministic features. Also, we extend the theoretical framework surrounding LOTOS for reasoning about systems described using this extended calculus.

7.1.0.4 Supporting the development of probabilistic systems

With respect to conformance, the development process for probabilistic systems includes two distinct activities: 'proving' and 'testing'. We *prove* that one PbLOTOS specification conforms to another PbLOTOS specification. We prove things about objects (PbLOTOS specifications) which exist in the world of mathematics. In contrast to the exact discipline of *proving* is the discipline of *testing*. We *test* that one real-world implementation conforms to a PbLOTOS specification. Unlike proving, testing involves uncertainties.

Later in this chapter, we develop an algorithm called SimChar upon which we base definitions of implementation relations. SimChar and the derived implementation relations can be used as a basis for 'proving' conformance between PbLOTOS specifications (see figure 7.1). Then we lay the foundations for a 'testing' framework for testing the conformance (as defined by the SimChar algorithm and derived implementation relations) between real-world probabilistic systems and PbLOTOS specifications.

Developing an implementation of a probabilistic system from a specification involves, amongst other aspects, the resolution of non-determinism (see section 4.4.2.5). An *implementation relation* formally expresses the notion of the *validity* of an implementation with respect to a specification. In order to support the development of probabilistic systems within a formal framework, we require that *implementation relations* validate probabilistic, as well as functional, aspects of implementations. Therefore, in this thesis we define implementation relations which support the development of probabilistic systems, from specifications which contain both non deterministic and probabilistic information. From such implementation relations we can derive notions of equivalence for systems exhibiting both probabilistic and non deterministic behaviour.

In practice, implementation relations are realised by sets of tests, where the observable responses of an implementation to the test suites are compared with the observable responses of the specification to the same test suites. In this thesis we lay foundations for a testing framework for probabilistic systems, supporting the implementation relations.

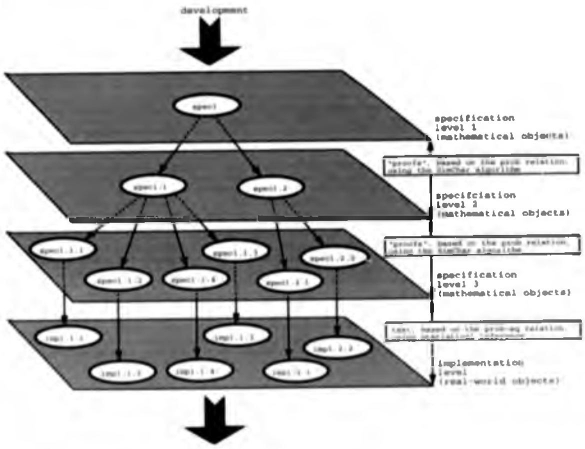


Figure 7.1: Developing probabilistic systems

7.2 Related work

Probabilistic models for process algebras have recently been studied by several researchers, e.g. [HJ89, HJ90, Han90, GJS90, vGSST90].

7.2.1 Reactive, generative and stratified probabilistic models

[vGSST90] present a three way classification of probabilistic models. They identify *reactive*, *generative* and *stratified* models. In all of these models, probabilities are associated with transitions.

In reactive models, the sum of the probability values for any set of alternative transitions with the same event must be 1 (or 0 if no such transitions exist). The reactive model does not relate the probabilities of different transitions. Like Milner [Mil80], van Glabbeek *et al.* characterize their models in terms of "button pushing experiments". For the reactive model, the "observer" may only attempt to press one button at a time. In the reactive model a button pushing experiment either succeeds with a probability of

1, or it fails. If the experiment succeeds, then the process makes an internal state transition with a probability defined by the probability distribution of the pressed button. For example, consider the reactive process A, shown in figure 7.2, given by:

$$A = \frac{1}{3}a.(c + d) + \frac{2}{3}a + b$$

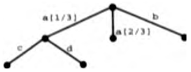


Figure 7.2: Reactive process A

Note that the sum of the probabilities for each action is 1, and that no information is given about the relative probability of performing an a transition compared to a b transition.

In generative models, the sum of the outgoing transitions from any one state must be 1 (if any outgoing transitions exist). Generative models relate the probabilities of the different outgoing transitions from any one state. The probabilities assigned to the outgoing transitions of any one state define the probability distribution when all possible transitions are offered. Consider the example of the generative process B, shown in figure 7.3, given by:

$$B = \frac{1}{4}a.(\frac{1}{2}c + \frac{1}{2}d) + \frac{2}{4}a + \frac{1}{4}b$$

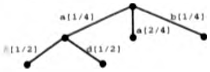


Figure 7.3: Generative process B

If the observer were allowed to attempt to press more than one button at a time, and attempted to press both a and b, a would occur with a probability of $\frac{3}{4}$ and b with a probability of $\frac{1}{4}$. In any "single button experiment", A and B cannot be distinguished.

Stratified models extend generative models with information on how to renormalize the probability distribution associated with a state, if some of the outgoing transitions from that state cannot be fired. This information follows the structure of the binary (probabilistic) choice operator. The following example will clarify this. Consider the example of the process $C1$ given by:

$$C1 = \frac{1}{3}a + \frac{1}{3}b + \frac{1}{3}c$$

Now, considering a restricted context in which transition b cannot be fired, we might expect, given the symmetry of $C1$, that $C1$ would "renormalize" to:

$$C1' = \frac{1}{2}a + \frac{1}{2}c$$

Using a stratified approach we can specify how "renormalization" is to occur. Consider the stratified $C2$ process, shown in figure 7.4, given by:

$$C2 = \frac{1}{3}a + \frac{2}{3}(\frac{1}{2}b + \frac{1}{2}c)$$

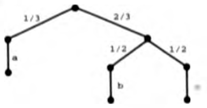


Figure 7.4: Stratified process $C2$

Now we can see how the nested probabilistic choice expressions can be used to structure the normalization information. If $C2$ were to be placed in a restricted context where b could not occur, then renormalization would yield the expression:

$$C2' = \frac{1}{3}a + \frac{2}{3}c$$

"Thus, in the stratified model, the intended relative frequencies are preserved in a level-wise fashion in the presence of restriction" [vGSST90]. However, we have still not dealt with the problem of *global normalization*, where, for example, the process $C2$ is synchronized with other processes which contain different probabilistic constraints on a, b and c.

Stratified models contain more information than generative models, and generative models contain more information than reactive models.

7.2.2 PCCS and the normalization function

The probabilistic models of [GJS90] and [vGSST90] are based on PCCS, a probabilistic version of Milner's SCCS [Mil83] in which SCCS expressions of the form $\sum_{i \in I} E_i$ are written as $\sum_{i \in I} [p_i] E_i$ (where p_i is the probability of of behaving like E_i). Two interesting aspects about PCCS (or any other probabilistic calculus) is how synchronous composition and restriction are dealt with. In PCCS, synchronous composition is interpreted as the simultaneous occurrence of independent events. Thus the synchronous composition $P \times Q$ of two processes, can behave as $P' \times Q'$ with a probability given by the product $\mu_1 \times \mu_2$, as shown in the equation below.

$$P - \alpha[\mu_1] \rightarrow P', Q - \beta[\mu_2] \rightarrow Q' \Rightarrow P \times Q - \alpha\beta[\mu_1 . \mu_2] \rightarrow p' \times Q'$$

A process is said to be *stochastic* if the sum of the probabilities of its derivations is 1. Else, if this sum is less than 1, the process is said to be *substochastic*. PCCS derivation schemas preserve the stochasticity of composite processes in a *restricted* way by employing a *normalization* function. When only a subset of the set of transition offers are fireable we call this restriction. (For example, process $C1$ above, is stochastic – the probabilities of its outgoing transitions sum to 1 – if transitions a, b and c are all fireable. However, in a restricted context, where b was not fireable then, without normalization, the process would be substochastic – the sum of its probabilities being $\frac{2}{3}$.) A substochastic process may deadlock ($C1$ in a restricted context, where b cannot be fired, has a probability of $\frac{1}{3}$ of deadlocking). The problem is that the use of restriction (and composition) results in substochastic processes (and hence, the non-zero possibilities of deadlock) as a consequence of the PCCS model. Such substochastic processes, a symptom of the PCCS model, are not reflected in the real world. We might say that the real world preserves stochasticity in the presence of restriction. This might lead to the conclusion that we need to introduce a "normalization" function into the PCCS model in order to arbitrate probability values in the presence of restriction and so preserve stochasticity. Indeed, this is the road taken in PCCS.

However, an alternative conclusion – the one taken in this thesis and, similarly, by [HJ89, HJ90] – is that the PCCS model is not minimalistic enough. We believe that the real world's preservation of stochasticity has more to do with the way in which we interpret real world behaviour. We think that 'the real world never deadlocks, something always happens next', but this is not a good reason to force our abstract models of the real world to 'always do something next' and never to deadlock. A deadlock in our abstract model may indicate that the system we are modelling can no longer behave within the bounds of our model world, while in the real world the system does something we do not expect or do not want it to do.

We believe that definition of explicit machinery (i.e. the normalization function) to negotiate probability distributions between synchronous processes is a symptom of an 'unnatural' probabilistic model. 'Unnatural' because it forces stochasticity within what is actually a '*restricted*' model world. Instead we propose a simpler model, where the model mechanisms make no attempt to preserve stochasticity, although the user may build systems which attempt to preserve stochasticity within themselves (sometimes called "reliable systems"). We present our simple probabilistic model in section 7.3, but for now return to the world of normalization.

Glabbeek *et al.* define the normalization function such that it makes stochastic non zero substochastic processes in the presence of restriction. The normalization function ς appears in the derivation rule for restriction thus:

$$P - \alpha[\mu_1] \rightarrow P' \;\Rightarrow\; P \upharpoonright A - \alpha[\mu/\varsigma(P,A)] \rightarrow P' \upharpoonright A$$

Basically, normalization calculates a probability distribution for the set of transitions fireable in the restricted context, given the probability distribution for the set of transitions fireable outside any restricted context and given some calculating formulae. The calculating formulae may perform normalization according to the stratified structure of the processes to be renormalized (as shown for processes $C2$ and $C2'$ above), or may be renormalized according to some other criteria.

7.2.3 Testing probabilistic processes

[LS89] explore the testing of probabilistic processes and define a testing algorithm which, with a probability of $1 - \epsilon$, where ϵ is arbitrarily small, can distinguish reactive processes which are not probabilistically bisimilar. Their processes are defined on a probabilistic transition system, in which the probability of a transition is either 0 or

$\geq \epsilon$. Thus all processes in their model are finitely branching (called "*image-finiteness*" in [HM85]) with $\frac{1}{\epsilon}$ the upper limit on the number of branches from any one state. Larsen and Skou then define a testing framework which they enhance to test properties written in Limited Modal Logic (LML) [BIM88], then the more expressive Hennessy-Milner Logic (HML) [HM85], and finally their own Probabilistic Modal Logic (PML). Hennessy and Milner [HM85] showed that if two processes satisfy exactly the same HML formulae, they are bisimilar. Similarly, Larsen and Skou describe how LML formulae and PML formulae can be ascribed the operational characterizations they call "$\frac{2}{3}$ bisimulation" and "probabilistic bisimulation" respectively. Also, their testing framework incorporates the notion of *hypothesis testing* at a *level of significance* (δ).

Larsen and Skou claim that their testing framework can test LML, HML or PML formulae against probabilistic processes, and hence distinguish processes which are not $\frac{2}{3}$ bisimilar, bisimilar or probabilistically bisimilar respectively. Probabilistic bisimilar is the limit of the distinguishing power of their testing framework, hence if two processes are probabilistically bisimilar then no test within their framework will distinguish them.

Larsen and Skou conclude by questioning their minimum probability assumption, and propose the ideas of "cost of a test" (a metric based on the number of basic experiments needed) and "informativeness of a test" (a metric which reflects the amount of information gained from a test) as issues for further study.

7.2.4 Probabilization

In [BM89], Bloom and Meyer show that if non-deterministic bounded branching processes P and Q are bisimilar, then there is an assignment of probabilities to the edges of the synchronization trees of P and Q, yielding processes P' and Q', such that P' and Q' are probabilistically bisimilar, and P' and Q' have the same probability of producing a given outcome under every test. Bloom and Meyer use the term "*probabilization*" to describe the act of assigning probabilities to the edges of the synchronization tree of a non-deterministic process, and to describe the probabilistic process resulting from the act of probabilization. Also, they touch on the idea of re-assigning, on recursion, probabilities to the edges of a recursive non-deterministic process. We expand on this idea later in this section.

7.2.5 A metric-space for the comparison of probabilistic processes

[GJS90] argue for the notion of a metric for measuring the similarity between probabilistic (PCCS) processes. In practice, a notion of an equivalence may be too restrictive, but a metric for the *distance* between probabilistic processes is likely to be more useful. This supports the appealing idea of making decisions based upon a quantitative comparison of the possibilities of failure between two functionally identical components, against their relative monetary costs. Giacalone *et al.* calculate the relative positions of (functionally identical) probabilistic processes within, what they term, a metric space. Then they say that process P can "safely" replace process P' if the distance between P and P', within the metric space, does not exceed (an arbitrary) ϵ.

7.2.6 Internal probabilistic choice and the alternating model

Hansson and Jonsson's timed, probabilistic calculus TPCCS [HJ90, Han90, HJ89] is defined on what they call the "alternating model". At each state, either a probabilistic or non-deterministic choice is made, and the model strictly alternates between probabilistic and non-deterministic states. Adoption of the alternation strategy allowed Hansson and Jonsson to structure the syntactic and semantic definitions of their TPCCS calculus into two halves. One half defines probabilistic aspects, and the other half defines non-deterministic and timing aspects. Also, the alternating model makes the definition of bisimulation equivalence neat.

In their calculus, Hansson and Jonsson define probabilistic transitions to be internal to processes, i.e. probability transitions occur without the influence of processes in the environment. Probabilities are not assigned directly to transitions which represent communication, since the occurrence of these transitions depends on the co-operation of the environment. The probabilistic choice operator is defined as a probability distribution over a set of possible successor states, reachable via internal transitions. In this respect, Hansson and Jonsson's work is similar to the work reported in this thesis.

Hansson and Jonsson take the branching time temporal logical CTL, and extend it with probability and quantitative time (to produce TPCTL). TPCTL can be used to formulate invariance, eventuality, precedence, reliability and performance properties. Hansson and Jonsson define a model checking algorithm for verifying if a TPCCS specification satisfies a TPCTL formula.

7.3 PbLOTOS: the formal framework

In this section we describe NP-LTSs, P-LTSs, PbLOTOS syntax and semantics, and additional notation which we use throughout this chapter.

We extend the definition of LTSs (Labelled Transition Systems) to define NP-LTSs (Non-deterministic and Probabilistic LTSs) and P-LTSs (Probabilistic LTSs). NP-LTSs are LTSs which may contain both non-deterministic and probabilistic transitions. P-LTSs are LTSs which contain only probabilistic transitions.

We use NP-LTSs as a semantic model for PbLOTOS (Probabilistic LOTOS). PbLOTOS is LOTOS enhanced by a small number of syntactic and semantic extensions which support probabilistic features. PbLOTOS has a probabilistic choice operator for specifying probability distributions over a set of internal probability transitions.

7.3.1 Definition of an NP-LTS

An NP-LTS (a *labelled transition system* (LTS) which may contain both *non-deterministic* and *probabilistic* transitions) is defined[1] as a 4-tuple: $\prec S, L \cup \{\mathbf{i}, \mathbf{p}_j\}, T, s_0 \succ$, where:

- S is an (enumerable) non-empty set of *states*;
- L is an (enumerable) set of *observable actions/events* or *label set*;

[1]by a straight-forward extension to the definition of an LTS as found in [ISO89b, BS86]

- **i** represents an *internal event*;
- $\mathbf{p}_j$ labels an indexed internal transition which has an associated probability of occurrence;
- $T = \{ -a \rightarrow \mid a \in L \cup \{\mathbf{i}\}\} \cup P$ is the set of *binary transition relations* on S;
- $P = \{ -\mathbf{p}_{j.1}(\mu) \rightarrow, -\mathbf{p}_{j.2}(1-\mu) \rightarrow \mid 0 < \mu < 1$, where j is such that $j.1$ and $j.2$ are, for any one state $s \in S$, unique indexes $\}$ is the set *of pairs* of binary, internal, *probabilistic transition relations* on S, and $\mu, 1-\mu$ is the probability distribution over the internal transitions labelled $\mathbf{p}_{j.1}, \mathbf{p}_{j.2}$;
- $s_0 \in S$ is the *initial state*

7.3.2 Definition of a P-LTS

A P-LTS is an NP-LTS which contains no non-deterministic transitions.
More formally, a P-LTS is an NP-LTS which satisfies[2]:

$$\forall s_k \in S \cdot ($$

$$((AllProbPairs(s_k) = 0) \wedge (AllHidden(s_k) = 0) \wedge (\forall \zeta \text{ in } L(s_k) \cdot (AllObsSingEv(s_k, \zeta) \leq 1)))$$

or

$$((AllObsAnyEv(s_k) = 0) \wedge (AllProbPairs(s_k) = 0) \wedge (AllHidden(s_k) \leq 1))$$

or

$$((AllObsAnyEv(s_k) = 0) \wedge (AllHidden(s_k) = 0) \wedge (AllProbPairs(s_k) \leq 1))$$

$$)$$

This says that a P-LTS is an NP-LTS which does not contain any states of the forms illustrated in section 7.4.3 figures 7.9 to 7.12 (or combinations of these).

7.3.3 Definition of PbLOTOS

A PbLOTOS behaviour expression is interpreted in terms of an NP-LTS, generated from the syntax of a PbLOTOS behaviour expression, by the axioms and inference schema rules[3] shown below.

[2]using the notation introduced and explained in section 7.3.1

[3]These rules are a straightforward extension to those found in [ISO89b, BS86].

Name	Syntax	Axioms or Inference Schema
inaction	**stop**	none
action-prefix		
unobservable	$\mathbf{i}; B$	$\mathbf{i}; B - \mathbf{i} \rightarrow B$
observable	$a; B$	$a \in L \Longrightarrow a; B - a \rightarrow B$
choice		
n-choice	$B1 [] B2$	$B1 - \alpha \rightarrow B1' \Longrightarrow B1 [] B2 - \alpha \rightarrow B1'$ $B2 - \alpha \rightarrow B2' \Longrightarrow B1 [] B2 - \alpha \rightarrow B2'$
p-choice	$B1 [= \mu] B2$	$B1 [= \mu]_j B2 - \mathbf{p}_{j,1}(\mu) \rightarrow B1$ $B1 [= \mu]_j B2 - \mathbf{p}_{j,2}(1 - \mu) \rightarrow B2$
parallel composition	$B1 \vert[a_1, \ldots a_n]\vert B2$	$B1 - \alpha \rightarrow B1', \alpha \notin \{a_1, \ldots a_n\} \Longrightarrow$ $B1 \vert[a_1, \ldots a_n]\vert B2 - \alpha \rightarrow B1' \vert[a_1, \ldots a_n]\vert B2$ $B2 - \alpha \rightarrow B2', \alpha \notin \{a_1, \ldots a_n\} \Longrightarrow$ $B1 \vert[a_1, \ldots a_n]\vert B2 - \alpha \rightarrow B2' \vert[a_1, \ldots a_n]\vert B2$ $B1 - a \rightarrow B1', B2 - a \rightarrow B2', a \in \{a_1, \ldots a_n\} \Longrightarrow$ $B1 \vert[a_1, \ldots a_n]\vert B2 - a \rightarrow B1' \vert[a_1, \ldots a_n]\vert B2'$
hiding	$B \backslash [a_1, \ldots a_n]$	$B - a \rightarrow B', a \in \{a_1, \ldots a_n\} \Longrightarrow$ $B \backslash [a_1, \ldots a_n] - \mathbf{i} \rightarrow B' \backslash [a_1, \ldots a_n]$ $B - \alpha \rightarrow B', \alpha \notin \{a_1, \ldots a_n\} \Longrightarrow$ $B \backslash [a_1, \ldots a_n] - \alpha \rightarrow B' \backslash [a_1, \ldots a_n]$

Note 1: The $[]$ and $[= \mu]$ operators are right-associative, such that $A [] B [= 0.4] C [] D [= 0.5] E$ is equivalent to (the parenthesized) $A [] (B [= 0.4] (C [] (D [= 0.5] E)))$.

Note 2: A μ term is a real-number, where $0 < \mu < 1$.

Note 3: For the convenience of the explanations within this chapter, we consider that each syntactic occurrence of the $[= \mu]$ operator within a PbLOTOS specification actually appears as $[= \mu]_j$, where j uniquely indexes a syntactic occurrence of $[= \mu]$ within the PbLOTOS specification. This is a justifiable convenience since we could have the "*flattening function #*" [ISO89b, section 7.3] (defined over the syntactic structure of a PbLOTOS specification) automatically perform this substitution. In other words, j indexes are neither part of the syntax nor the semantics of PbLOTOS but of the static semantics.

Note 4: The above axioms and inference schemas define extensions to Basic LOTOS which yield 'Basic PbLOTOS'. Extensions to full LOTOS have not been defined because this would result in detailed definitions which are unnecessary to explain the essence of the PbLOTOS definition. The two main parts of PbLOTOS which are alien to LOTOS are the addition of a definition for probabilistic transitions (section 7.3.1) and the inference schema for *p-choice* expressions (above).

7.3.4 PbLOTOS examples

7.3.4.1 Example 1

P [=0.6] Q

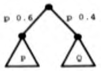


Figure 7.5: Tree notation for example 1

Figure 7.5 illustrates how the above PbLOTOS expression is interpreted. The **p** branches represent probabilistic transtions. A **p** transition is an internal transition, similar to an **i** transition but attributed with a probability value. The probability value of a branch is specified by a probabilistic choice operator $[= \mu]$, where the left branch has a probability value μ (where $0 < \mu < 1$) and the right branch has a probability value $1 - \mu$.

7.3.4.2 Example 2

a;b;**stop** [=0.6] a;(c;**stop** [=0.75] b;**stop**)

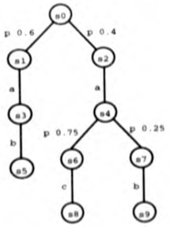


Figure 7.6: Tree notation for example 2

The probability of taking a path from state x to state y is the product of the probability values found on all **p** branches between states x and y. For example 2:

$$P(\text{path between states } s_0 \text{ and } s_5) = 0.6$$
$$P(\text{path between states } s_0 \text{ and } s_9) = 0.4 \times 0.25 = 0.1$$
$$P(\text{path between states } s_2 \text{ and } s_8) = 0.75$$

The probability of performing a particular transition sequence from a state x, is the sum of the probabilities of all paths from x which exactly include the necessary transition sequence. For example 2:

$$\begin{aligned} P(= ab \Rightarrow |s_0) &= P(\text{path between states } s_0 \text{ and } s_4) \\ &\quad + P(\text{path between states } s_0 \text{ and } s_9) \\ &= (0.6) + (0.4 \times 0.25) \\ &= 0.7 \end{aligned}$$

Where $P(= ab \Rightarrow |s_0)$ means 'the probability of performing the transition sequence $= ab \Rightarrow$ given state s_0 (from state s_0)'.

7.3.4.3 Example 3

(a;**stop** [=0.3] b;**stop**) || (a;**stop** [=0.9] b;**stop**)

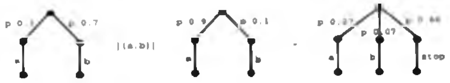


Figure 7.7: Tree notation for example 3

Figure 7.7 illustrates the effective result of a simple synchronous combination of probabilistic PbLOTOS expressions.

The synchronous combination contains a branch which leads to deadlock. This is because (like [HJ90], and unlike [vGSST90, GJS90]) our PbLOTOS model does not implicitly preserve stochasticity. Instead, the PbLOTOS specifier may build systems which explicitly attempt to preserve stochasticity within themselves (sometimes known as "reliable systems"). In this way PbLOTOS makes "reliability" an explicit design issue. We believe that models which implicitly preserve stochasticity by using a normalization function (e.g. [vGSST90, GJS90]) are 'unnatural' (see section 7.2.2).

7.3.4.4 Example 4

(a;**stop** []b;**stop**) || (a;**stop** [=0.9] b;**stop**)

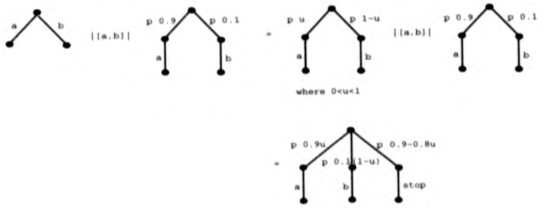


Figure 7.8: Tree notation for example 4

Figure 7.8 illustrates the effective result of a simple synchronous combination of an non-deterministic expression and a probabilistic expression. The reason for the assignment of the probability values μ and $1 - \mu$ to the left and right branches of the first, non-deterministic tree, is explained in the sections which follow.

7.3.5 Trace-refusal notation

Below we recall the trace-refusal notation used by [BS86].

Notation	**Interpretation**
L	alphabet of observable actions
L^*	set of strings over L
$\sigma, \sigma_1, \ldots, \varepsilon$	$\in L^*$, with ε denoting the empty string
$a, b, c, a_1, a_2, \ldots$	$\in L$
$\alpha, \alpha_1, \ldots$	$\in L \cup \{\mathbf{i}, \mathbf{p}_j\}$
$P, Q, P_1, P_2, S_1, I, \ldots$	$\in S$
$P - \alpha_1 \ldots \alpha_n \rightarrow Q$	$\exists P_i (0 \leq i \leq n) \cdot P = P_0 - \alpha_1 \rightarrow P_1 - \alpha_2 \rightarrow \ldots - \alpha_n \rightarrow P_n = Q$
$P = \varepsilon \Rightarrow Q$	$P = Q$ or $\exists (n \geq 1$ and $\hbar \in \{\mathbf{i}, \mathbf{p}_j\}) \cdot P - \hbar^n \rightarrow Q$
$P = a \Rightarrow Q$	$\exists P_1, P_2 \cdot P = \varepsilon \Rightarrow P_1 = a \Rightarrow P_2 = \varepsilon \Rightarrow Q$
$P = \sigma \Rightarrow Q$	if $\sigma = a_1 \ldots a_n$ then
	$\exists P_i (0 \leq i \leq n) \cdot P = P_0 = a_1 \Rightarrow P_1 = a_2 \Rightarrow \ldots = a_n \Rightarrow P_n = Q$
	also denoted as $P = a_1 \ldots a_n \Rightarrow Q$
$P = \sigma \Rightarrow$	$\exists Q \cdot P = \sigma \Rightarrow Q$
$P \neq \sigma \Rightarrow$	$\neg(P = \sigma \Rightarrow)$
$Tr(P)$	$\{\sigma \in L^* \mid P = \sigma \Rightarrow\}$
$S_1, S_2, S_3, \ldots$	used to denote N&P-LTSs
$I_1, I_2, I_3, \ldots$	used to denote P-LTSs

An NP-LTS state may be identified with a PbLOTOS process, where the state is interpreted as the initial state of the process. Throughout this chapter we use the words *state* and *process* interchangeably.

7.4 Implementation relations for PbLOTOS systems

We consider that a probabilistic specification (an NP-LTS) describes a set of probabilistic implementations (P-LTSs). Then, upon this basis, we define an implementation relation (a pre-order), called *probabilization*, for NP-LTSs. The probabilization relation can be used as a conformance relation in the development of PbLOTOS systems.

7.4.1 Non-deterministic branching as probabilistic branching

This work assumes that non-deterministic branching may be considered as probabilistic branching. This assumption is justifiable if we consider that an NP-LTS specification describes a set of real world implementations, and that real world systems display probabilistic behaviours.

We adopt Bloom and Meyer's term *probabilization* [BM89], and use the phrase *probabilization of a system* to mean that all non-deterministic branching within the system is viewed as probabilistic branching.

Before providing further explanation of *probabilization*, we define some additional notation on states and transitions.

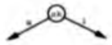
7.4.2 States and transitions notation

Notation	**Interpretation**
$AllProbPairs(s_k)$	$= \{\prec -\mathbf{p}_{j,1} \rightarrow, -\mathbf{p}_{j,2} \rightarrow \succ \mid -\mathbf{p}_{j,1} \rightarrow, -\mathbf{p}_{j,2} \rightarrow \in s_k \times S\}$
i.e. set of all pairs of probabilistic transitions from a state $s_k \in S$	
$AllHidden(s_k)$	$= \{-\mathbf{i} \rightarrow \mid -\mathbf{i} \rightarrow \in s_k \times S\}$
i.e. set of all internal $\mathbf{i}$ event transitions from a state $s_k \in S$	
$AllObsSingEv(s_k, a)$	$= \{-a \rightarrow \mid -a \rightarrow \in s_k \times S\}$
i.e. set of all observable a event ($a \in L$) transitions from a state $s_k \in S$	
$AllObsAnyEv(s_k)$	$= \{-a \rightarrow \mid -a \rightarrow \in s_k \times S, a \in L\}$
i.e. set of all observable event transitions from a state $s_k \in S$	
$\lvert AllObsSingEv(s_k, a)\rvert$	= the cardinality of the set $AllObsAnyEv(s_k)$

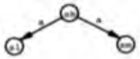
7.4.3 The occurrence of non-deterministic branching

In an NP-LTS, non-deterministic branching occurs at a state s_k if:

$|AllHidden(s_k)| > 0$ or $|AllObsSingEv(s_k, a)| > 1$

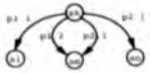


$a; \ldots [] i; \ldots$
Figure 7.9

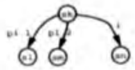


$a; L [] a; M$
Figure 7.10

$|AllProbPairs(s_k, a)| > 1$ or $(|AllHidden(s_k)| > 0) \wedge (|AllProbPairs(s_k)| > 0)$



$(L[=\mu_1]M)[](M[=\mu_2]N$
Figure 7.11



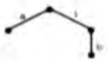
$(L[=\mu_1]M)[]N$
Figure 7.12

and, of course, combinations of these scenarios. Note that the above scenarios are examples of the cases excluded by the definition of a P-LTS in section 7.3.2.

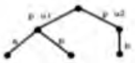
7.4.4 Example probabilizations of non-deterministic branchings

We may replace a state where non-deterministic branching occurs, by a state with appropriate probabilistic branching. Consider the following example substitutions:

7.4.4.1 Example probabilization 1 (figures 7.13, 7.14)



is probabilized as



where

$0 < \mu_1 < 1$
$0 < \mu_2 < 1$
$\mu_1 + \mu_2 = 1$

The example above illustrates one of the simplest cases of probabilization. In the NP-LTS in figure 7.13 non-determinism is caused by the fact that both an observable transition (a) and an unobservable transition (i) originate from the same state (the LTS does not satisfy the predicate in section 7.3.2). In order to turn this NP-LTS into a P-LTS the probabilization operation must remove this non-determinism. The probabilization operation also must preserve the observable properties of the NP-LTS (e.g. $P(= b \Rightarrow) = 1$).

The probabilization operation uses the fact that the expression:

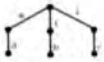
$$a;\mathbf{stop}[]\mathbf{i};b;\mathbf{stop} \tag{7.1}$$

is observationally equivalent ($\approx_{te}$, [BS86]) to the expression:

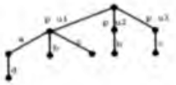
$$\mathbf{i};(a;\mathbf{stop}[]b;\mathbf{stop})[]\mathbf{i};b;\mathbf{stop} \tag{7.2}$$

This resolves the problem of observable and unobservable transitions originating from the same state. The probabilization operation replaces the two **i** transitions in expression 7.2 by a pair of **p** (probability) transitions. This removes the non-determinism. A b transition can be found on both branches of the probability pair so b can occur with a probability of 1. This preserves an observable property of the original NP-LTS. (See appendix H for a more detailed explanation).

7.4.4.2 Example probabilization 2 (figures 7.15, 7.16)



is probabilized as

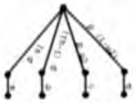


where:

$0 < \mu_1 < 1$
$0 < \mu_2 < 1$
$0 < \mu_3 < 1$
$\mu_1 + \mu_2 + \mu_3 = 1$

In the NP-LTS in figure 7.15 non-determinism arises as a result of an observable transition and two unobservable transitions originating from the same state. This example of non determinism is more complicated than in the previous example (figure 7.13), but it can be resolved using the same principle.

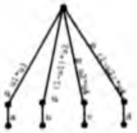
7.4.4.3 Example probabilization 3 (figures 7.17, 7.18)



where

$0 < \mu_1 < 1$
$0 < \mu_2 < 1$

is probabilized as

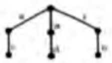


where:

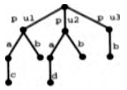
$0 < \mu_1 < 1$
$0 < \mu_2 < 1$
$0 < \mu_3 < 1$
$0 < \mu_4 < 1$
$\mu_3 + \mu_4 = 1$

In the NP LTS in figure 7.17 non determinism arises as a result of two pairs of probability transitions originating from the same state.

7.4.4.4 Example probabilization 4 (figures 7.19, 7.20)



is probabilized as



where:

$$0 < \mu_1 < 1$$
$$0 < \mu_2 < 1$$
$$0 < \mu_3 < 1$$
$$\mu_1 + \mu_2 + \mu 3 = 1$$

In the NP-LTS in figure 7.19 non-determinism arises as a result of two observable transitions and one unobservable transition originating from the same state.

Sections 7.4.5 and 7.4.7 and appendix H define and explain an algorithm for probabilizing NP-LTSs as P-LTSs, in the same fashion as shown by the above examples.

7.4.5 Characterization of an NP-LTS as a set of possible P-LTSs

We consider that an NP-LTS S_1 implicitly defines a set of *probabilistic implementations* (P-LTSs), i.e.:

$$S_1 = \{I \mid I \ \underline{prob}\ S_1, I \text{ is a P-LTS}\}$$

We characterize the set of probabilistic implementations of S_1 by a *set of simultaneous equations* (*SimChar*[4]). *SimChar* describes *each* possible P-LTS implementation of S_1, and enumerates the probabilities of all observable traces of *each* P-LTS implementation.

In general, there will be a set of solutions to each *SimChar*. Each solution in such a *solution set* will describe one possible probabilistic implementation (P-LTS) of the NP-LTS.

We structure *SimChar* as a *set of trace probabilities* (*trprob*) and a *set of auxiliary equations* (*auxeq*), i.e.:

$$SimChar = \prec trprob, auxeq \succ$$

The *set of trace probabilities* is a set of pairs, each pair consists of (and is identified by) an observable trace of S_1 (ranged over by σ_i) and a *free-term*[5] (ranged over by μ_i) that represents the probability of the trace, i.e.:[6]

$$trprob = \{\prec \Leftarrow \sigma_i \Rightarrow, \mu_i \succ \mid \sigma_i \in Tr(S_1), \mu_i = \ldots\}$$

The *set of auxiliary equations* consists of equations relating free-terms and ground-terms, e.g.:

$$auxeq = \{0 < \mu_i < 1, 0 < \mu_j < 1, \mu_i + \mu_j = 1\}$$

[4]'simultaneous equational characterization'

[5]free-term is synonymous with the term μ-term, used elsewhere in this thesis

[6]see the trace-refusal notation 7.3.5.

An example will help clarify the above prose. Consider the PbLOTOS process Q_3, illustrated as an NP-LTS:

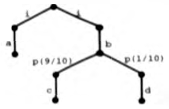


Figure 7.21: Process Q_3

The *set of trace probabilities* of Q_3 is:

$$\{ \quad \prec\prec a \succ, \mu_1 \succ, \quad \prec\prec b, c \succ, \tfrac{9}{10}\mu_2 \succ, \quad \prec\prec b, d \succ, \tfrac{1}{10}\mu_2 \succ \quad \}$$

In effect, the **i** branches of Q_3 have been labelled with μ_1 and μ_2.
The *set of auxiliary equations* of Q_3 is:

$$\{ \quad 0 < \mu_1 < 1, \quad 0 < \mu_2 < 1, \quad \mu_1 + \mu_2 = 1 \quad \}$$

The *set of simultaneous equations* which characterize Q_3 as a set of possible P-LTSs is the union of the *set of trace probabilities* and the *set of associated equations* of Q_3.

The *solution set* for Q_3 is represented diagrammatically in figure 7.22. Each point on the $\mu_1 + \mu_2 = 1$ line represents a *solution* where μ_1 and μ_2 are valued by the vertical and horizontal axis coordinate values (respectively) of this point.

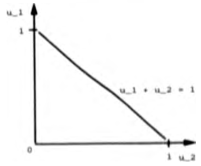


Figure 7.22: Solution set for Q_3

7.4.6 A probabilization relation

An *implementation relation* is often used to formalize the notion of 'a *valid* implementation of a formal specification' [BS86, Led91a, Led90]. Implementation relations are often pre-orders. This thesis defines the pre-order $\underline{prob}$ as a formal implementation relation between NP-LTSs.

"Probabilization" [BM89] of an NP-LTS involves replacing non-deterministic transitions by probabilistic transitions. We consider that an NP-LTS S implicitly defines a set of implementations $\{I|I\ \underline{prob}\ S\}$ of P-LTSs. (We use I to range over P-LTSs, and use S to range over NP-LTSs. P-LTSs are also NP-LTSs by definition, but not vice versa).

We characterize the set of probabilistic implementations $\{I|I\ \underline{prob}\ S\}$ as a set of simultaneous equations, where each equation describes the probability of an observable trace of S. The *free variables*[7] within the simultaneous equations generate the *set* of probabilistic implementations. These *free variables* are used to range over the possible probabilizations of non-deterministic branches in S.

We say that for any two NP-LTSs, S_1 and S_2, $S_1\ \underline{prob}\ S_2$ *iff* S_1 describes only a non-strict subset of probabilistic implementations which S_2 describes, i.e. $\{I|I\ \underline{prob}\ S_1\} \subseteq \{I|I\ \underline{prob}\ S_2\}$. Furthermore, S_1 and S_2 are *probabilization equivalent*, written $S_1\ \underline{prob{-}eq}\ S_2$, *iff* $\{I|I\ \underline{prob}\ S_1\} = \{I|I\ \underline{prob}\ S_2\}$.

In previous subsections we enhanced LOTOS with internal probabilistic branching, thus providing a convenient means for generating NP-LTSs. Our aim now is to provide a characterization of an NP-LTS as a set of simultaneous equations ($SimChar$), which will allow us to provide an operational definition of the $\underline{prob}$ and $\underline{prob{-}eq}$ relations. Later, we suggest a simple statistical testing framework for establishing whether a probabilistic system (P-LTS) I is a valid implementation of an NP-LTS S (i.e. $I\ \underline{prob}\ S$).

7.4.7 Definitions for the characterization of an NP-LTS as a set of simultaneous equations

We define the *simultaneous equational characterization* $SimChar$ of an NP-LTS S_1 by:

$$SimChar(s_0, 0, d_{max}, 0, \emptyset)$$

where s_0 is the initial state of the NP-LTS S_1,
d_{max} is the maximum observable trace depth,
the other parameters are explained in the following text.

The definition of the function $SimChar$ is given as an algorithm. Before we define it we introduce more notation:

The following table provides an explanation of some of the notation (in addition to that which has already been defined) used in the algorithmic definition of $SimChar$.

[7]also called free-terms or μ-terms elsewhere in this thesis

Notation	Interpretations
s_0	is assumed to be the initial state of the system S.
s_k	is the state of the system S, with which the particular instance of $SimChar$ was instantiated.
$s_\_$	the bar subscript indicates that $s_\_$ can be identified with any one state s_k in S.
$= \sigma \Rightarrow$	this is shorthand for saying $s_0 = \sigma \Rightarrow s_\_$
$\mu_{x.i.m}$	used to index free-terms associated with i transitions, where $m = 1..AllHidden(s_k)$.
$\mu_{x.o.\zeta.n}$	used to index free-terms associated with observable ζ event transitions, where $n = 1..AllObsSingEv(s_k, \zeta), \zeta \in L(s_k)$.
$\mu_{x.p.l}$	used to index free-terms associated with pairs of probabilistic transitions, where $m = 1..AllProbPairs(s_k)$.
$\mu_{x.frac}$	used to denoted the free-term associated with all observable event transitions — this is the 'fraction' that the probability values of all observable transitions from any one state must sum to.
$\oplus, \bigoplus$	concatenates $auxeq$ terms
$\uplus, \biguplus$	$trset$ union: for $trsets$, member identification is based on the trace fields of member pairs. When two or more members have identical trace fields (ignoring the 'to-state'), then a $\uplus$ union of their sets will result in a set with a member with a trace field identical to that of the members in question, and a probability field formed from the sum of the probability fields of these members. For example,

$$
\begin{aligned}
&\{ \prec s_0 = \sigma_1 \Rightarrow s_k, \mu_k \succ, \prec s_0 = \sigma_2 \Rightarrow s_g, \mu_{g1} \succ \} \\
\uplus\ &\\
&\{ \prec s_0 = \sigma_1 \Rightarrow s_e, \mu_e \succ, \prec s_0 = \sigma_3 \Rightarrow s_g, \mu_{g2} \succ \} \\
=\ &\\
&\{ \prec s_0 = \sigma_1 \Rightarrow s_\_, (\mu_k + \mu_g) \succ, \prec s_0 = \sigma_2 \Rightarrow s_g, (\mu_{g1}) \succ, \\
&\quad \prec s_0 = \sigma_3 \Rightarrow s_g, (\mu_{g2}) \succ \}
\end{aligned}
$$

Note how the pairs $\prec s_0 = \sigma_1 \Rightarrow s_k, \mu_k \succ$ and $\prec s_0 = \sigma_1 \Rightarrow s_e, \mu_e \succ$ have been identified as 'trace field equivalent (ignoring the 'to-states')' and have been unified. Whereas the pairs $\prec s_0 = \sigma_2 \Rightarrow s_g, \mu_{g1} \succ$ and $\prec s_0 = \sigma_3 \Rightarrow s_g, \mu_{g2} \succ$ are not identified as trace field equivalent because $\sigma_2 \neq \sigma_3$.

$\overset{\oplus}{\uplus}, \overset{\bigoplus}{\biguplus}$	unifies $\prec trprob, auxeq \succ$ pairs, such that,

$$
\begin{aligned}
&\prec trprob1, auxeq1 \succ \overset{\oplus}{\uplus} \prec trprob2, auxeq2 \succ \\
=\ &\\
&\prec trprob1 \uplus trprob2, auxeq1 \oplus auxeq2 \succ
\end{aligned}
$$

The following algorithm provides the definition of $SimChar$:

```
SimChar($s_k$, $x$, $d_{max}$, $d_{curr}$, trprob) is

    (* $s_k \in S$ the current state
       $x$ is a unique index given to this instantiation
       $d_{max}$ is the maximum observable trace depth to which SimChar recurses
       $d_{curr}$ is the current depth
       trprob carries the set of trace probabilities that have
              been accumulated so far in this branch of
              recursive instantiations of SimChar
    *)
```

```
if $|AllTrans(s_k)| = 0$ then
      (* no further transitions possible *)
      return($\prec trprob, \emptyset \succ$)

elseif $d_{curr} = d_{max}$ then
      (* maximum depth reached *)
      return($\prec trprob, \emptyset \succ$)

else
      (* there are transitions from $s_k$ *)

      (* now create a new subset of auxeq for the auxiliary equations to
       * be associated with the state $s_k$... *)

      $auxeq' := \emptyset$

      if $|AllObsAnyEv(s_k)| = 0$ then
            (* no direct observable trans from $s_k$ *)
            $auxeq' := auxeq' \oplus \{\mu_{x.frac} = 0\}$

      elseif $|AllObsAnyEv(s_k)| > 0$ then
            (* direct observable trans from $s_k$ exist *)

            if $|AllHidden(s_k)| > 0$ or $|AllProbPairs(s_k)| > 0$ then
                  (* direct i or p trans from $s_k$ exist *)
                  $auxeq' := auxeq' \oplus \{0 < \mu_{x.frac} < 1\}$

            elseif $|AllHidden(s_k)| = 0$ and $|AllProbPairs(s_k)| = 0$ then
                  (* no direct i or p trans from $s_k$ *)
                  $auxeq' := auxeq' \oplus \{\mu_{x.frac} = 1\}$

            endif

            $auxeq' := auxeq' \oplus \{\forall \zeta \in L(s_k) \cdot (\sum_{n=1..|AllObsSingEv(s_k,\zeta)|} \mu_{x.o.\zeta.n} = \mu_{x.frac})\}$

      endif

      $auxeq' := auxeq' \oplus \{$
                        $\forall m = 1..|AllHidden(s_k)| \cdot 0 < \mu_{x.i.m} < 1,$
                        $\forall j = 1..|AllProbPairs(s_k)| \cdot 0 < \mu_{x.p.j} < 1,$
                        $\forall \zeta \in L(s_k) \cdot \forall n = 1..|AllObsSingEv(s_k,\zeta)| \cdot 0 < \mu_{x.o.\zeta.n} < 1,$
                        (     $\mu_{x.frac}$
                           $+ \sum_{m=1..|AllHidden(s_k)|} \mu_{x.i.m}$
                           $+ \sum_{j=1..|AllProbPairs(s_k)|} \mu_{x.p.j}$
                        ) = 1
                        $\}$

      (* now launch more SimChar instantiations to trace through all the states which
       * follow $s_k$, and then unify the trprob sets, and auxeq sets, which are
       * returned after these recursing instantiations rewind *)


      $\prec trprob', auxeq'' \succ :=$
                  (
                        (* recurse to follow all i trans... *)
            $(\overset{\oplus}{\biguplus}_{\forall s_k - i_m \rightarrow s_q \in AllHidden(s_k)}$
                            $SimChar(s_q, x.i.m, d_{max}, d_{curr}, trprob'')$
            where $trprob'' = \{\prec s_0 = \sigma \Rightarrow s_k = x \Rightarrow s_q, \mu_k \times (\mu_{x\,i\,m} + \mu_{x\,frac}) \succ\}$
            )
         $\overset{\oplus}{\uplus}$
            (* recurse to follow all p trans pairs... *)
            $(\overset{\oplus}{\biguplus}_{\forall s_k - p_j [\mu_{j\,1}] \rightarrow s_{q\,j\,1}, s_k - p_j [(1-\mu_{j\,1})] \rightarrow s_{q\,j\,2} \in AllProbPairs(s_k)}$ (
                            (* follow j.1 p trans... *)
                            $SimChar(s_{q\,j\,1}, x.p.j.1, d_{max}, d_{curr}, trprob'')$
                         $\overset{\oplus}{\uplus}$
                            (* follow j.2 p trans... *)
                            $SimChar(s_{q\,j\,2}, x.p.j.2, d_{max}, d_{curr}, trprob''')$
                         )
            where $trprob'' = \{\prec s_0 = \sigma \Rightarrow s_k = x \Rightarrow s_{q\,j\,1}, \mu_k \times \mu_{x\,p\,j} \times (\mu_{j\,1} + \mu_{x\,frac}) \succ\}$
                  $trprob''' = \{\prec s_0 = \sigma \Rightarrow s_k = x \Rightarrow s_{q\,j\,2}, \mu_k \times \mu_{x\,p\,j} \times ((1 - \mu_{j\,1}) + \mu_{x\,frac}) \succ\}$
            )
         $\overset{\oplus}{\uplus}$
            (* recurse to follow all observ event trans... *)
            $(\overset{\oplus}{\biguplus}_{\forall \zeta \in L(s_k)}$ (
                       (* follow all trans for a given event $\zeta$... *)
                       $(\overset{\oplus}{\biguplus}_{\forall s_k - \zeta_n \rightarrow s_q \in AllNonSingleEv(s_k, \zeta)}$
                                      $SimChar(s_q, x.o.\zeta.n, d_{max}, (d_{curr} + 1), trprob'')$
                       where $trprob'' = \{\prec s_0 = \sigma \Rightarrow s_k = \zeta \Rightarrow s_q, \mu_k \times \mu_{x\,o\,\zeta\,n} \succ\}$
                       )
                     )
         )
      where $trprob = \{\prec s_0 = \sigma \Rightarrow s_k, \mu_k \succ\}$

      (* and finally return the trace probability set $trprob'$, and the auxiliary equation
       * set associated with state $s_k$ ($auxeq'$) unified with the auxiliary equation
       * set associated with the states which follow $s_k$ ($auxeq''$) *)
      return($trprob'$, $auxeq' \oplus auxeq''$)

   endif

end (* SimChar *)
```

For any NP-LTS S_1, we define the function *SimChar* which generates the ***set of simultaneous equations*** which characterize S_1, in the style explained in section 7.4.5. Basically, the *SimChar* function takes an NP-LTS, recurses through the traces of the NP-LTS, treats non-deterministic transitions as probabilistic transitions with free-term probabilities, constructs the ***set of auxiliary equations*** (*auxeq*) which limits these free-terms, and constructs the ***set of trace probabilities*** (*tract*). The set of simultaneous equations produced by *SimChar*, in effect, defines a P-LTS whose probability transitions are assigned free terms (with values limited within the solution set of the simultaneous equation).

Appendix H provides an example application of the *SimChar* algorithm to a simple PbLOTOS system. The example illustrates how *SimChar* probabilizes an NP-LTS which contains one of the non-deterministic branching scenarios described in section 7.4.3. Appendix II provides a step-by-step guide through the instantiations of *SimChar*, illustrating how *SimChar* produces the set of simultaneous equations which characterizes an NP-LTS as a P-LTS, and highlighting important points about the algorithm's method.

7.4.8 The recursive assignment of μ terms in SimChar

For recursive NP-LTSs, the *SimChar* algorithm assigns new free terms (μ probability values) to non-deterministic transitions on each recursion. If we visit the same state twice, the second visit will result in the assignment of μ terms to the transitions from that state, different from the previously assigned μ terms to these transitions. The consequence is that this definition of *SimChar* makes more identifications than if it were to recognise re-visits to states and re-use the previous μ term assignments.

An example should clarify what we have said. Consider the NP-LTS in figure 7.23.

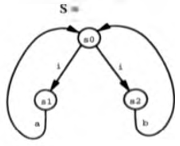


Figure 7.23: A recursing NP-LTS

The definition of *SimChar* gives[8]:

$$
\begin{aligned}
&SimChar(s_0, 0, 2, 0, \emptyset) = \\
&\qquad \prec \{ \prec = a = b \Rightarrow, \mu_{0.i.1} \times \mu_{0.i.1.i.2} \succ, \\
&\qquad\quad \prec = b = a \Rightarrow, \mu_{0.i.2} \times \mu_{0.i.2.i.1} \succ, \\
&\qquad\quad \prec = a = a \Rightarrow, \mu_{0.i.1} \times \mu_{0.i.1.i.1} \succ, \\
&\qquad\quad \prec = b = b \Rightarrow, \mu_{0.i.2} \times \mu_{0.i.2.i.2} \succ \\
&\qquad \}, \\
&\qquad \{ 0 < \mu_{0.i.1} < 1, 0 < \mu_{0.i.2} < 1, \mu_{0.i.1} + \mu_{0.i.2} = 1, \\
&\qquad\quad 0 < \mu_{0.i.1.i.1} < 1, 0 < \mu_{0.i.1.i.2} < 1, \mu_{0.i.1.i.1} + \mu_{0.i.1.i.2} = 1, \\
&\qquad\quad 0 < \mu_{0.i.2.i.1} < 1, 0 < \mu_{0.i.2.i.2} < 1, \mu_{0.i.2.i.1} + \mu_{0.i.2.i.2} = 1 \\
&\qquad \}, \\
&\quad \succ
\end{aligned}
$$

Whereas, *SimChar'* modified to re-use μ value assignments when re-visiting states and transitions on recursion, would give:

$$
\begin{aligned}
&SimChar'(s_0, 0, 2, 0, \emptyset) = \\
&\qquad \prec \{ \prec = a = b \Rightarrow, \mu_{0.i.1} \times \mu_{0.i.2} \succ, \\
&\qquad\quad \prec = b = a \Rightarrow, \mu_{0.i.2} \times \mu_{0.i.1} \succ,
\end{aligned}
$$

[8]The following trace probability sets (*trprobs*) and auxiliary equation sets (*auxeqs*) have been rationalized for the sake of space and clarity. They are not exact reproductions of the output from *SimChar*.

$\prec\!\!= a = a \Rightarrow, \mu_{0.1} * \mu_{0.1} \succ,$
$\prec\!\!= b = b \Rightarrow, \mu_{0.2} * \mu_{0.2} \succ$
$\}$,
$\{\ 0 < \mu_{0.1} < 1, 0 < \mu_{0.2} < 1, \mu_{0.1} + \mu_{0.2} = 1$
$\}$,
$\succ$

Now also consider the P-LTS implementation in figure 7.24. The trace probabilities for implementation I are:

$\{\quad \prec\!\!= a = b \Rightarrow, [illegible] \succ,$
$\prec\!\!= b = a \Rightarrow, \frac{1}{40} \succ,$
$\prec\!\!= a = a \Rightarrow, [illegible] \succ,$
$\prec\!\!= b = b \Rightarrow, [illegible] \succ \quad \}$

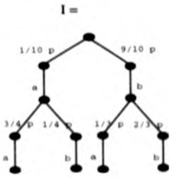


Figure 7.24: A P-LTS implementation

Now, with the unmodified definition of *SimChar*, we find that we can solve the simultaneous equations given by $SimChar(s_0, 0, 2, 0, \emptyset)$ (involving the free terms: $\mu_{0.1}, \mu_{0.2}, \mu_{0.1}, \mu_{0.1.0.1}, \mu_{0.1.0.2} \mu_{0.2.0.1}, \mu_{0.2.0.2}$) such that I <u>*prob*</u> S.

Whereas, with the modified version of *SimChar* (*SimChar'*), no solution to the simultaneous equations given by $SimChar'(s_0, 0, 2, 0, \emptyset)$ (involving the free terms: $\mu_{0.1}, \mu_{0.2}$) can be found which identify I as a valid probabilization of S. Therefore, using *SimChar'*, we would have to conclude that I ~~*prob*~~ S.

We have chosen to adopt the *SimChar* algorithm (and not the modified *SimChar'*) as our basis for defining probablistic implementation relations. This is because *SimChar* identifies a larger number of valid implementations in the face of unknown mechanisms which are represented by non deterministic choice.

7.4.9 The avoidance of infinite recursion in SimChar

There are two related issues here: infinite looping involving observable transitions, and infinite looping involving only hidden transitions.

7.4.9.1 Infinite looping involving observable transitions

SimChar avoids the problem of infinitely looping sequences which include observable transitions by requiring the user to specify a maximum observable trace depth.

This practical restriction on the *SimChar* algorithm has repercussions for the definitions of probabilistic implementation relations based on it. We define probabilistic relations between PbLOTOS specifications as relations between the *SimChar* characterizations of the PbLOTOS specifications:

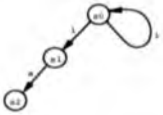
$$SimChar(sys1_0, 0, d1_{max}, 0, \emptyset) \;\; prob\_relation \;\; SimChar(sys2_0, 0, d2_{max}, 0, \emptyset)$$

The $d1_{max}$ and $d2_{max}$ values ought to be such that all the information gathered by the two invocations of *SimChar* to the trace depths $d1_{max}$ and $d2_{max}$ is all the information that is necessary to decide whether or not the relation *prob_relation* holds. For instance, say we know that both systems *sys1* and *sys2* completely unfold all their unique behaviours at an observable trace depth of n, and then simply recurse. Then it may be sufficient to set both $d1_{max}$ and $d2_{max}$ to the value n. This issue has a lot in common with the issue of satisfactorily testing a probabilistic implementation. In section 7.5 we discuss how finite tests can be used to check possibly infinite implementations to arbitrary confidence levels.

For the remainder of this chapter, when giving examples of the probabilistic implementation relations between systems, we assume that the maximum observable trace depth parameters of the involved *SimChar* instances have been set to appropriate values.

7.4.9.2 Infinite looping involving only hidden transitions

The given definition of *SimChar* fails to avoid the problem of infinitely looping sequences of internal (unobservable) transitions. The maximum observable trace depth restriction does not apply to looping sequences of transitions containing only hidden (i) transitions. Let us consider the problem using the example NP-LTS in figure 7.25 (and the accompanying PbLOTOS text which generates it).

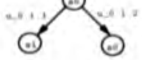


generated by

```
process S[a] :noexit :=
i; a; ...
[]
i; S[a]
endproc
```

Figure 7.25: An example with an infinite i-loop

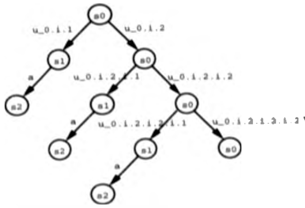
Now, if we diagrammatically depict the traces involved in the first instantiation of $SimChar(s_0, 0, 1, 0, \emptyset)$ we get figure 7.26:



where $P(\Rightarrow a \Rightarrow) = \mu_{0\,i\,1}$
where $\mu_{0\,i\,1} + \mu_{0\,i\,2} = 1$

Figure 7.26: The first instantiation of *SimChar*

Successive instantiations of *SimChar* will produce figure 7.27:



where

$P(= a \Rightarrow) =$

$\mu_{0.i.1}$
$+(\mu_{0.i.2} \times \mu_{0.i.2.i.1})$
$+(\mu_{0.i.2} \times \mu_{0.i.2.i.2} \times \mu_{0.i.2.i.2.i.1})$
$+ \ldots$

where

$\mu_{0.i.1} + \mu_{0.i.2}(\mu_{0.i.2.i.1} +$
$\mu_{0.i.2.i.2}(\mu_{0.i.2.i.2.i.1} +$
$\mu_{0.i.2.i.2.i.2}(\ldots))) = 1$

Figure 7.27: Successive instantiations of *SimChar*

Hence, as the number of i-loops $\rightarrow \infty$, $P(= a \Rightarrow) \rightarrow 1$. Obviously it is not practically possible to compute $P(= a \Rightarrow)$ using *SimChar*, and the present definition of *SimChar* would endlessly recurse on encountering an infinite i-loop. Two possible solutions are:

1. Modify the *SimChar* algorithm so that it can detect infinite i-loop scenarios, and take appropriate finite action. For example, the modified algorithm would detect the i-loop scenario discussed above and assign $P(= a \Rightarrow)$ the value 1.

2. Parameterize *SimChar* with the maximum number of successive i transitions it can trace before aborting (in a similar fashion as for the maximum observable trace depth parameter).

Neither of these modifications is vital for the work described in this chapter. Therefore, to keep definitions simple and clear, we ignore the existence of the i-loop problem; none of the remaining examples involving *SimChar* will contain infinite i-loops.

7.4.10 An implementation relation and associated equivalence

An ***implementation relation*** formally expresses the notion of ***validity*** with respect to a specification [BS86]. An implementation relation is not necessarily symmetric. This reflects the directed, asymmetric nature of the development process, in which an implementation can validly replace a specification but not vice versa.

[Led91a] defines the relation $\underline{imp}$ as the reference implementation relation. $\underline{imp}$ may be instantiated as a number of more specific implementation relations — the obvious examples being $\underline{conf}$, $\underline{red}$, $\underline{ext}$, etc. (see [BS86]). $\underline{imp}$ is reflexive (a specification being a valid implementation of itself), but not necessarily transitive (indicating that an implementation may not be used as an intermediate specification, e.g. the $\underline{conf}$ relation).

$\underline{imp\text{-}eq}$ is the equivalence based on $\underline{imp}$. The following definition is taken from [Led91a].

7.4.10.1 Definition of an implementation relation

S_1 *imp−eq* $S_2 \iff \{I | I \text{ } imp \text{ } S_1\} = \{I | I \text{ } imp \text{ } S_2\}$,
where $\{I | I \text{ } imp \text{ } S_i\}$ denotes the set of processes which are valid implementations of the specification S_i according to the relation *imp*.

Informally, two specifications are *imp−eq* *iff* they describe exactly the same set of valid implementations in accordance with *imp*.

7.4.11 A probabilization relation, and associated equivalence

We adopt Bloom and Meyer's term *probabilization* [BM89], and formally define this notion for NP LTSs. The *prob* relation is a particular, transitive instance of *imp*, and therefore a pre-order (i.e. reflexive and transitive) relation.

Pre-orders are well suited as *implementation relations*. They define an ordering among systems which reflects their relative positions along the 'development trajectory'. If $S_1 < S_2$ according to such an ordering, then S_1 is a valid implementation of S_2 (by some criterion). The criterion formally expressed by the *prob* pre-order, is the probabilization of non-deterministic branching (i.e. the replacement of non-deterministic choices by probabilistic choices). Also, S_1 may itself be used as an intermediate specification, due to the transitive character of *prob*.

7.4.11.1 Definition of a probabilization relation

First some additional notation:

Notation	Interpretation
$tract_{S_1}$	the *tract* generated by applying *SimChar* to S_1.
$auxeq_{S_1}$	the *auxeq* generated by applying *SimChar* to S_1.
μ_{S_1} bindings	ground-terms bound to S_1 free-terms, such that each free-term is bound/associated with one ground-term.
μ_{S_1} bindings $\models auxeq_{S_1}$	replacing all free-terms found in $auxeq_{S_1}$ by the ground-terms associated with the free-terms by μ_{S_1} bindings satisfies the equations found in $auxeq_{S_1}$.
$Prob_{S_1}(\sigma)$	the probability of a trace σ, by the process S_1, given the replacement of free-terms in $auxeq_{S_1}$ by ground-terms

Let P_1 and P_2 be PbLOTOS processes. Then
P_2 *prob* P_1 *iff*

(i) $Tr(P_2) = Tr(P_1)$

(ii) $\forall(\mu_{P_2}$ bindings $\models auxeq_{P_2}) \cdot \exists(\mu_{P_1}$ bindings $\models auxeq_{P_1}) \, \forall \sigma \in Tr(P_1) \cdot Prob_{P_2}(\sigma) = Prob_{P_1}(\sigma)$

Informally, P_2 is a probabilization of P_1 *iff*

(i) the trace sets of P_1 and P_2 are equal, and

(ii) it is always possible to find solutions to $auxeq_{P_1}$ such that the probabilities of P_1 and P_2 traces are identical, for all possible solutions to $auxeq_{P_2}$.

An alternative informal definition is: P_2 is a probabilization of P_1 *iff* P_2 describes a subset of the probabilistic implementations (P-LTSs) which P_2 describes.

7.4.11.2 Definition of probabilization equivalence

Let P_1 and P_2 be PbLOTOS processes. Then
P_2 $\underline{prob\!-\!eq}$ P_1 *iff*

(i) $Tr(P_2) = Tr(P_1)$

(ii) $\forall(\mu_{P_2}$ bindings $\models auxeq_{P_2}) \cdot \exists(\mu_{P_1}$ bindings $\models auxeq_{P_1}) \cdot \forall\sigma \in Tr(P_1) \cdot Prob_{P_2}(\sigma) = Prob_{P_1}(\sigma)$

(iii) $\forall(\mu_{P_1}$ bindings $\models auxeq_{P_1}) \cdot \exists(\mu_{P_2}$ bindings $\models auxeq_{P_2}) \cdot \forall\sigma \in Tr(P_1) \cdot Prob_{P_2}(\sigma) = Prob_{P_1}(\sigma)$

Informally, P_1 and P_2 are probabilization equivalent *iff* P_1 and P_2 both describe exactly the same set of the probabilistic implementations (P-LTSs).
An alternative formal definition is: P_2 $\underline{prob\!-\!eq}$ P_1 *iff*

(i) P_2 $\underline{prob}$ P_1

(ii) P_1 $\underline{prob}$ P_2

7.4.11.3 Examples of the probabilization relations

Figure 7.28 portrays a family of specifications and the prob relations. The trace probability sets (*trprobs*) and auxiliary equation sets (*auxeqs*) for these specifications are not given in exact *SimChar* notation (e.g. using free-terms with indexes such as $\mu_{0.frac}$, $\mu_{0.o.e.1}$, etc.) for the sake of space. However, the notation scheme used should be fairly self evident, and the reader should be able to attain a overview of how the ideas of NP-LTS, *SimChar*, probabilization, etc. are interrelated.

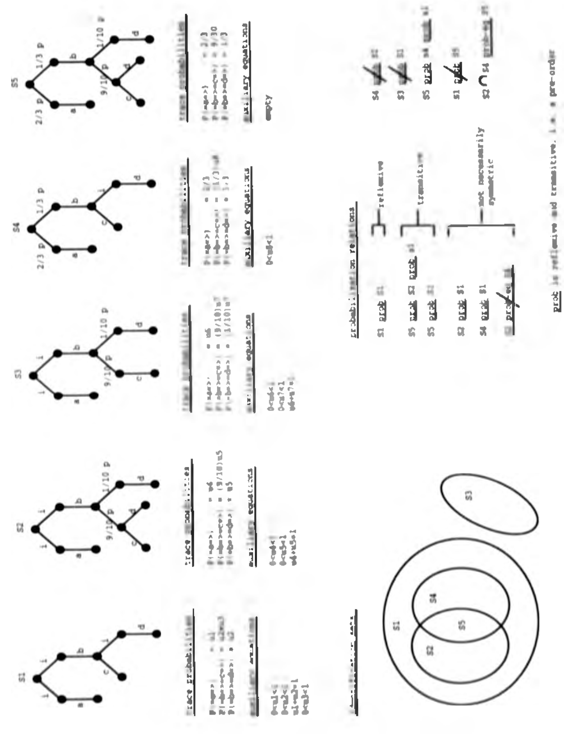


Figure 7.28: A family of specifications and their prob relations

7.4.12 Discussion

In contrast to other probabilistic process algebras, we have moved the emphasis away from the semantics of the PbLOTOS language, and instead placed many of the 'probabilistic concepts' in the associated theory of relations. The consequence of this is that the probabilistic aspects of the PbLOTOS semantics are simpler than the probabilistic aspects of the semantics of other process algebra, but PbLOTOS theory for relations is, consequently, more complex.

For example, "normalization" is an instance of a 'probabilistic concept' which we have located in PbLOTOS's theory of relations but which, in contrast, has been located within the language semantics of other process algebras, such as PCCS [vGSST90]. Section 7.2.2 describes the "normalization" function that is built into the semantics of PCCS. Its task is to preserve stochasticity. For PbLOTOS, *auxeq* (section 7.4.5) performs, in effect[9], the same function as PCCS's "normalization" function. However, *auxeq* is defined as an aspect of the *SimChar* algorithm which exists as a part of PbLOTOS's theory of relations, and not as an actual part of the PbLOTOS language semantics.

The consequence of locating many of the 'probabilistic concepts' within the theory of relations, rather than within the language semantics, is that a PbLOTOS specification is not completely meaningful unless interpreted within the framework of the PbLOTOS theory of relations. We see this as a perfectly natural situation since, analogously, no real world behaviour is, in itself, meaningful unless interpreted within some framework of understanding. Moreover, this results in greater flexibility. For instance, in the next section we examine how to test PbLOTOS implementations using statistical methods. Changing the test statistic used for conformance, in effect changes the type of probability distribution (e.g. unimodal, multimodal, etc.) that we expect the probabilistic behaviour to follow. We might not have this flexibility if we had somehow built the probability distribution type into the language semantics. Also, it seems more appropriate to separate information such as the probability distribution type from the actual specification. A more appropriate place for such information is within a "conformance testing framework" (see section7.5.4).

7.5 Testing real world implementations against PbLOTOS specifications

We suggest a simple statistical testing framework for establishing whether a probabilistic implementation (a P-LTS) is a valid implementation of a probabilistic specification (an NP-LTS), according to the probabilization relation.

[9] The equations contained in *auxeq* ensure that individual probabilistic transitions are assigned values such that the whole system remains stochastic.

7.5.1 PbLOTOS needs a framework of testing theory

The probabilistic aspects of PbLOTOS's syntax and semantics, described in section 7.3, are meaningless unless interpreted within a framework of testing theory. Section 7.3 describes how values μ denoting probability, are associated with transitions within P-LTSs or NP-LTSs. But these values only become meaningful when interpreted within a statistical testing framework. Sections 7.4.5 to 7.4.11 have developed implementation relations for use within this testing framework, and this subsection concentrates on applying these implementation relations.

7.5.2 Testing 'valid refinement'

In practice, we want to be able to test whether a particular implementation is a valid refinement of a given specification. The notion of 'valid refinement' is often expressed as a set of implementation relations. In essence, a set of implementation relations will describe a set of properties which the implementation must satisfy in order to be a 'valid refinement' of the specification. Thus, in general, we would like to be able to test if a given implementation satisfies a set of given properties.

7.5.3 Testability

A test is a finite exercise, whereas the behaviour of a system may be infinite (or not wholly contained within the attention of the test). In general this implies that testing cannot be completely conclusive, but instead testing establishes if a particular system satisfies a particular property to some *confidence level*.

Complete confidence is usually only achievable in the world of mathematics (where we "prove" results), or (in the real world) for *"liveness-properties"* in the case where the property is observed within the test[10]. In the real world, complete confidence is the exception rather than the norm. Normally we attribute to the result of a test a confidence level within the range *completely no confidence* to *complete confidence*.

The point that we want to emphasis is that correctness, and testing for correctness, are not black and white issues. Now, if we argue for the probabilistic nature of real world systems, the consequence is that testing is not merely an exercise in boolean logic, but in statistical inference. Then, it follows that the probabilistic information needed to drive the statistical aspects of tests ought to be present somewhere within the specification or a 'conformance testing framework'.

7.5.4 Using a conformance testing framework for a test

The conformance testing framework includes system requirements additional to those in the specification. The framework includes information identifying conformance points, testing practices, conformance environments, conformance assumptions (including, for probabilistic systems, expected types of probability distributions), etc. (see [Hog90]).

[10] e.g. for the property 'a light will switch on at least once', a test observation may show that the light has indeed switched on.

Test construction must be done with respect to the specification, the conformance testing framework and the implementation under test. For example, we test if an implementation *imp* satisfies a property *prty* (expressed by the specification), in conjunction with additional assumed properties *add_prty* (expressed by the conformance framework), in a restricted environment *res_env* (expressed by the conformance framework); i.e.:

$$imp \wedge res\_env \models prty \wedge add\_prty$$

We may find the (probabilistic) information needed to construct (statistical) tests from the specification or conformance testing framework, or both.

Aside: Before proceeding we make the following observation about this area of work.

The vocabulary used in the area of specification, testing and conformance can be confusing and lead to disagreement. For example, one commentator might talk about "testing if an implementation satisfies properties" while another might talk about "testing if an implementation, within the restricted context described by the conformance testing framework, satisfies, to some confidence level, properties of the specification". Then the second commentator may ridicule the first, inferring that the first commentator's view of testing is too simplistic. In reality though, the first commentator's vocabulary may differ from the second's, and may be saying the same thing in a more concise way. However both commentators may recognise the same essential ingredients for testing even if their vocabularies differ.

Bear this in mind when, later, we write: $imp \models prty$ (rather than something like: $imp \wedge res\_env \models prty \wedge add\_prty$).

7.5.5 Hypothesis testing

Normally the more tests a system passes, the greater the confidence we have in the correctness of the system. The logical extension of this is to suppose that we can test a system to confirm its correctness with an arbitrary level of confidence. Testing, to an arbitrary level of confidence, that a system possesses some property is normally known as hypothesis testing in statistics [Kay93, CC83, Fel68, DH70].

A specification implicitly defines a set of properties that a valid implementation must satisfy. We write $sys \models prty$ to indicate that a system *sys* satisfies a property *prty* of a specification.

Now, if we want to perform a test to establish if a system satisfies a property, we may form a *null hypothesis* and an *alternative hypothesis* as follows:

H_0: $sys \models prty$
H_1: $sys \not\models prty$

Thus we have reduced our test to a "decision problem" — we must, given the evidence contained in our test observations, decide for H_0 or H_1.

The null hypothesis H_0 is a statement of our base belief, while the alternative hypothesis H_1 is the statement for which we will attempt to accumulate supporting evidence.

This formulation of hypotheses is not without question. The reader might ask why the hypotheses were not formulated the other way around. After all, one might argue that the *base* assumption, H_0, should say that, in general, any given system will not be a valid implementation of a particular specification. And then H_1 should state what we want to 'prove' (i.e. that the system is indeed valid). Actually, the formation of hypotheses is very sensitive to exactly what it is we are trying to prove, and to test observations. Thus, formation of hypotheses will be peculiar to a particular test situation.

Given the "decision problem" we require a "decision rule". Therefore we partition the "sample space" (observation space) into two regions: the "acceptance region" (AR) and the "rejection region" (RR). The decision rule is that if the test observations fall within RR then reject H_0, else if the test observations fall within AR then do not reject H_0.

The property *prty* is the "test statistic" – it helps us differentiate H_0 from H_1. *prty* ought to be a specification of all the possible behaviours in the (sub)system under test.

In hypothesis testing, we can set various parameters to values to provide quantitative indications of the confidence with which we can accept the result of the test. Setting these various parameters allows us to bias our decision rule.

If, for example, we take the hypothesis formulated as shown above. If we are a prospective buyer of the system under test, we will want to accept the null hypothesis H_0 only if we are very sure that a system satisfies the specification (i.e. a stringent "quality control"). Or, stated from a different angle, we will want to decide in favour of the null hypothesis only if we are very sure that the results of tests on the system strongly indicate freedom from implementation *errors*. On the other hand, if we are the producer of the system under test, we will want to reject the null hypothesis H_0 only if we are very sure that a system does not satisfy the specification.

In statistical hypothesis testing we will never know whether or not our decision is really correct. The following table, taken from [Kay93], shows the four possible decision categories.

		Reality	
		H_0 True	H_1 True
Decision	Decide for H_0	✓	Type II error ×
	Decide for H_1	Type I error ×	✓

- A Type I error occurs when we reject H_0, when in reality it is true.
- A Type II error occurs when we accept H_0, when in reality it is false.

Returning to the example above, in the rôle of system buyer, our interest would lie in reducing Type II errors, i.e. we would want to reduce the chances of accepting a false H_0 (an erroneous system). However, in the rôle of system producer, our interest would lie in reducing Type I errors, i.e. we would want to reduce the chances of rejecting a true H_0 (a correct system).

So how do we bias a decision rule, or assess the level of confidence that we should attribute to a test result (i.e. to a 'decision')? The parameters that statisticians use for this purpose are:

$$\alpha = Pr(Type\ I\ error) \text{ and } \beta = Pr(Type\ II\ error)$$

In order to reduce Type I errors we should set α (often known as the "significance level") to a low value. Unfortunately, as we reduce the risk of making a Type I error, we increase the risk of making a Type II, and vice versa.

Another metric for measuring how much confidence we should place in a test result is the "Power" of the test procedure.

$$\text{high Power} = \text{low Pr(Type II error)}$$

Say our decision is to accept H_0. Then, if the Power is high, we may feel confident in this decision. Otherwise, if the Power is low, or the sample size is small, maybe we should gather more evidence.

Statistics provides a multiplicity of theories, formulae and advice for hypothesis testing. The intention here was just to introduce statistics as a framework within which to perform conformance testing for probabilistic systems. This is the extent of our explanation of statistics for conformance testing. Examples later in this section adopt and use statistical methods.

7.5.6 Properties for test

Below we list some of the properties that we might wish to test for in PbLOTOS specifications.

In section 7.4.5 we showed how to characterize a PbLOTOS system as a set (*SimChar*) of observable traces and probabilities. We use observable trace and probability notation to express examples in the list of properties below.

Eventuality properties: properties that eventually will become true, e.g. $\prec = a \Rightarrow, 1 \succ$, event a will eventually happen.

Reliability properties: properties that are true with a specified probability, e.g. $\prec = Successful\_Send \Rightarrow, \frac{99}{100} \succ$, a (event sequence) *Successful_Send* will be 99% reliable. (This is particularly relevant for PbLOTOS systems.)

Performance properties: properties that with a specified probability become true within some specified time, e.g. $\prec = task\_finished\{setInterval(3,9)\} \Rightarrow, \frac{7}{10} \succ$, *task_finished* will, with a probability of $\frac{7}{10}$, occur within the time interval 3..9. (This is particularly relevant for combined TLOTOS and PbLOTOS systems.)

In this list, we have considered just those properties relevant to PbLOTOS. For a list of properties expressible in full XL see section 8.2.3.4. For the remainder of this chapter, we restrict ourselves to considering only eventuality and reliability properties.

7.5.7 Formulating a test: an example

Suppose that we have developed an implementation and we want to test if it satisfies a specification, i.e.:

H_0: $imp \models spec$
H_1: $imp \not\models spec$
How do we go about this?

7.5.7.1 Establishing the hypotheses

Firstly, we must decide what the satisfies relation $\models$ means. For the example developed in this section, we take $\models$ (above) to mean $\underline{prob{-}eq}$ — this requires that the implementation preserves the probabilistic properties of the specification. Also, for this example, we take S5 in figure 7.28 as the specification. Hence, as our *general hypotheses* we have:

H_0: $imp\ \underline{prob{-}eq}\ spec$
H_1: $imp\ \underline{prob\!\!\not{-}eq}\ spec$

Our general approach to testing these hypotheses is to characterize the specification S5 by a set of properties, and then test that the implementation satisfies each of these properties. We accept H_0 only if the implementation satisfies all the properties from the set, otherwise we reject H_0. Each property will form a sub-hypothesis, and we will test each of these sub-hypotheses separately. The separate testing of properties leads to some redundancy but we ignore this for this simple example.

Section 7.4.7 showed that we can characterize any PbLOTOS specification by (SimChar) a set of trace probabilities and a set of auxiliary equations. From figure 7.28 we see that, for S5, the set of auxiliary equations is empty[11], and the set of trace probabilities is:

$$\{ \quad \prec\prec a \succ, \tfrac{2}{3} \succ, \\ \prec\prec b, c \succ, \tfrac{9}{30} \succ, \\ \prec\prec b, d \succ, \tfrac{1}{30} \succ \quad \}$$

We view each trace probability pair as a property to be satisfied by the implementation, and so pose these (below) as sub-hypotheses.

$H_{1.0}$: $imp \models \prec\prec a \succ, \frac{2}{3} \succ$ $\quad H_{2.0}$: $imp \models \prec\prec b, c \succ, \frac{9}{30} \succ$ $\quad H_{3.0}$: $imp \models \prec\prec b, d \succ, \frac{1}{30} \succ$
$H_{1.1}$: $imp \not\models \prec\prec a \succ, \frac{2}{3} \succ$ $\quad H_{2.1}$: $imp \not\models \prec\prec b, c \succ, \frac{9}{30} \succ$ $\quad H_{3.1}$: $imp \not\models \prec\prec b, d \succ, \frac{1}{30} \succ$

7.5.7.2 Which statistical inference methods?

We have decomposed our conformance problem and established our hypothesis, but on the basis of which statistical inference methods do we perform testing? In other words, which statistical methods should we use to interpret test observations, and decide for one hypothesis or another? First, we attempt to develop a statistical method based upon some assumptions about properties enjoyed by discrete event systems. However, the complexity of real world systems pervades this approach, and we abandon it in favour of an established statistical method (χ^2).

[11]or, more accurately (in respect of the SimChar definition), has one solution set of values for free variables, and these values have been substituted directly into the set of trace probabilities.

Statistical inference based on special assumptions Now we try to develop a method for interpreting test observations, based upon some assumptions special to the discrete event nature of most distributed computing systems.

Assumption 1: The "minimum probability assumption" says that, within a PbLOTOS system, no probability transition has a value less than Δ. This assumption seems plausible for certain systems; implementation details may be available which describe what internal transitions exist within in the system, their nature, and minimum chance of occurring (in particular environments).

The minimum probability assumption implies a finite limit on probabilistic branching from any one state. In fact $\frac{1}{\Delta}$ is the universal upper limit on such branching (see figure 7.29). This is similar to "image-finiteness" in [HM85].

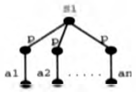


Figure 7.29: "image-finiteness" for PbLOTOS

$$\begin{array}{l} P(a_1) \geq \Delta \\ P(a_2) \geq \Delta \\ \dots \\ P(a_n) \geq \Delta \\ P(a_1) + P(a_2) + \cdots + P(a_n) = 1 \\ \Rightarrow n \leq \frac{1}{\Delta} \end{array}$$

Assumption 2: The "copying assumption" says that we can make a copy of a system at any state (see [Abr87]). This would allow us to re-run trials of a test several times, each time on a 'fresh copy' of the system. Initially this seems plausible because it can often be achieved in practice, e.g. for software systems by core dumping. Later we air reservations about this assumption.

The implications of this assumption, for the example in figure 7.29, are that we can compute the number of test trials that we need to perform on (fresh copies of) the system (at state S_1) in order to achieve a particular probability of achieving evidence of any one particular transaction.

Let us put these two assumptions to work. Say we would like to test, for the system shown in figure 7.29, the hypotheses:

H_0: $sys \models = a_1 \Rightarrow$
H_1: $sys \not\models = a_1 \Rightarrow$

Our decision rule will be:

Accept H_0 if test evidence falls within AR
Reject H_0 if test evidence falls within RR

We choose quantitative values v_{AR} and $1 - v_{AR}$ for the areas AR and RR (see figure 7.30), where $0 < v_{AR} < 1$. (A higher v_{AR} value corresponds to a lower α value.)

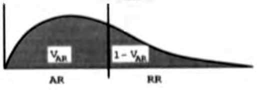


Figure 7.30: AR and RR

If, in reality, H_0 is true, we want our test to show this. This implies that we need to construct our test so that the probability of obtaining evidence for H_0 is $\geq v_{AR}$ when H_0 is really true, i.e.

P(evidence for H_0 when H_0 is really true) $\geq v_{AR}$

Assumptions 1 and 2 suggest that it is possible to construct such a test. Consider figure 7.31, which is the system in figure 7.29 except that the transitions $a_2 \ldots a_n$ have been collapsed into a single transition b.

Using only assumptions 1 and 2, we can say that:

$$\begin{aligned} P(\text{evidence of an } a_1 \text{ transition}) &= P(S_1 = a_1 \Rightarrow) \\ &\geq \Delta \\ &= 1 - (1 - \Delta) \end{aligned}$$

for one trial.

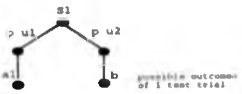


Figure 7.31: Testing using one trial

If we run a second independent trial using a fresh copy of the system (see figure 7.32), then we can say that:

$$\begin{aligned} P(\text{evidence of an } a_1 \text{ transition}) &= P(S_1 = a_1 \Rightarrow |S_1 = a_1 \Rightarrow) \\ &\quad + P(S_1 = b \Rightarrow |S_1 = a_1 \Rightarrow) \\ &\quad + P(S_1 = a_1 \Rightarrow |S_1 = b \Rightarrow) \\ &\geq \Delta \times \Delta \\ &\quad + (1 - \Delta) \times \Delta \\ &\quad + \Delta \times (1 - \Delta) \\ &= 2\Delta - \Delta^2 \\ &= 1 - (1 - \Delta)^2 \end{aligned}$$

for two independent trials.

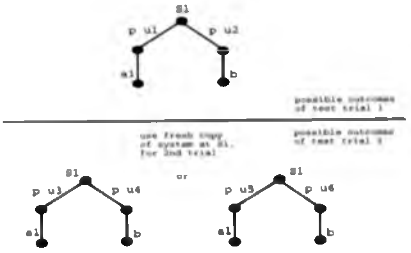

S1
p u1
p u2
a1
b
possible outcomes of test trial 1
use fresh copy of system at S1, for 2nd trial
possible outcomes of test trial 2
S1
p u3
p u4
a1
b
or
S1
p u5
p u6
a1
b

Figure 7.32: Testing using two trials

This result generalizes to:

P(evidence of an a_1 transition) $\geq 1 - (1 - \Delta)^n$
for n independent trials.

Thus, by specifying n (the number of test trials) it would seem that we can arbitrarily govern the probability of observing evidence for a transition. Therefore, returning to the issue of how to construct our test such that:

P(evidence for H_0, when H_0 is really true) $\geq v_{AR}$

and using the above result, we have:

$$\begin{aligned}
& P(\text{evidence for } H_0\text{, when } H_0 \text{ is really true}) \geq v_{AR} \\
\Rightarrow\ & P(\text{evidence of an } a_1 \text{ transition}) \geq v_{AR} \\
\Rightarrow\ & 1 - (1 - \Delta)^n \geq v_{AR}
\end{aligned}$$

This result tells us that we can choose the size of AR (and hence the chance of a Type I error), and then choose the number of test trials such that supporting evidence for H_0 will be found if H_0 is true in reality. This result seems too good to be true. Below we give two difficulties which make this result unusable.

Difficulty 1: This testing model is too simple to check the actual probability of a transition. It only detects the fact that a transition may possibly occur, if in reality it can. It may be possible to extend the testing model to check actual probabilities, but there is another problem:

Difficulty 2: In real systems, not all transitions and states are observable. We can only detect observable events, but there may be a hidden number of unobservable

> internal transitions and states between observable events. Therefore, although it may be possible to make a copy of a system at a state between any two observable events, we cannot be sure at which exact state we have made the copy. This, in itself is not the problem. The problem is that there may now exist an indeterminable number of hidden states between observable transitions, and hence an indeterminable amount of branching between observable transitions (even with the minimum probability assumption). This invalidates our previous result. One 'solution' would be to assume a maximum limit on the number of internal states between any two observable events, which would mean that, again, we could compute the upper limit on branching between observable events. However, we think that this assumption contradicts what we our trying to do: test the observable behaviour of systems without having to disassemble them.

Therefore, we abandon this as a generally applicable approach for test inference, although it might be usable for systems with strictly controlled, and determinable, numbers of states. Instead we turn to an established statistical inference method χ^2.

Statistical inference based on χ^2 χ^2 is often known as the "goodness of fit" statistic. The χ^2 test is not directed against any specific alternative from the hypotheses, but provides a quantitative indication of how well test observations fit each alternative. This fits well with our objective, so we adopt χ^2 as a method for interpreting test observations.

First, we briefly introduce χ^2 as a method for interpreting test observations. Consider again the example in figure 7.29, and the hypotheses we posed for it:

H_0: $sys \models = a_1 \Rightarrow$
H_1: $sys \not\models = a_1 \Rightarrow$

These are formulated as a χ^2 test table as:

outcome	expected	observed
$= a_1 \Rightarrow$	$\mu_{exp} N$	$\mu_{obs} N$
$\neq a_1 \Rightarrow$	$(1 - \mu_{exp})N$	$(1 - \mu_{obs})N$
	N	N

where:
N is the number of test trials
$d.f.$ (degrees of freedom) $= (row - 1)(col - 1) = (2 - 1)(2 - 1) = 1$

$$\chi^2_{obs} = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

where:
O_i is the observed frequency in cell i
E_i is the expected frequency in cell i
n is the number of outcomes $= row = 2$

Reject H_0 if $\chi^2_{obs} > \chi^2_{\alpha,d.f}$

If we are the producer of the implementation then, most likely, we will want to reject

H_0 only if we have substantial evidence to support H_1.
$\Rightarrow$ reduce Type I errors $\Rightarrow$ set a low α value.

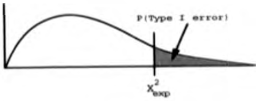


Figure 7.33: Type I error

We may decide that we require $P\{reality \in RR | H_0\ true\} \leq 0.05$, i.e. $P\{Type\ I\ error\} \leq 0.05$. Then, from tables [Fel68] we find that $\chi^2_{0.05,1} = 3.84$. So we reject H_0 if $\chi^2_{obs} > 3.84$

7.5.7.3 Example application

We have decided to use the χ^2 statistic to interpret test observations and decide for one hypothesis or another, and have introduced this statistic. Now we return to the example of testing an implementation against specification S5, and apply χ^2.

To make the testing procedure more clear, we cast the implementation under test as the concrete example represented in figure 7.34.

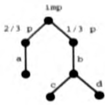


Figure 7.34: The implementation under test

Now we formulate the sub-hypotheses as χ^2 tests:

Sub-hypothesis 1

outcome	expected	observed
$\equiv a \Rightarrow$	N	N
$\not\equiv a \Rightarrow$	N	N
	N	N

$H_{1.0}$: $imp \models \prec\prec a \succ, \frac{2}{3} \succ$
$H_{1.1}$: $imp \not\models \prec\prec a \succ, \frac{2}{3} \succ$

$$\chi^2_{obs} = \sum \frac{(O-E)^2}{E} = \frac{(\frac{2}{3}N - \frac{2}{3}N)^2}{\frac{2}{3}N} + \frac{(\frac{1}{3}N - \frac{1}{3}N)^2}{\frac{1}{3}N} = 0$$

Therefore $\neg(\chi^2_{obs} > 3.84) \Rightarrow$ accept H_0
The test observations for the implementation in figure 7.34, would lead us to make this decision for any number N of test trials ≥ 1.

Sub-hypothesis 2

outcome	expected	observed
$= bc \Rightarrow$	$\frac{9}{30}N$	$\frac{10}{30}N$
$\neq bc \Rightarrow$	$\frac{21}{30}N$	$\frac{20}{30}N$
	N	N

$H_{2.0}$: $imp \models \prec\prec b, c \succ, \frac{9}{30} \succ$
$H_{2.1}$: $imp \not\models \prec\prec b, c \succ, \frac{9}{30} \succ$

$$\chi^2_{obs} = \sum \frac{(O-E)^2}{E} = \frac{(\frac{10}{30}N - \frac{9}{30}N)^2}{\frac{10}{30}N} + \frac{(\frac{20}{30}N - \frac{21}{30}N)^2}{\frac{21}{30}N} \approx 0.0053N$$

Now, if we have carried out ≥ 726 test trials (i.e. $N \geq 726$) then $\chi^2_{obs} > 3.84 \Rightarrow$ reject H_0, however, if we have carried out < 726 test trials (i.e. $N < 726$) then $\neg(\chi^2_{obs} > 3.84)$ $\Rightarrow$ accept H_0.

In other words, we would need to carry out at least 726 test trails before obtaining a sufficient amount of evidence (at the chosen α level) to reject H_0. This raises the question of, in general, how do we decide the number of test trials that should be carried out? The answer to this depends on two related issues: what we know about the expected behaviour of the system under test, and the test statistic.

The first of these issues takes us back to our discussions on the "minimum probability assumption" and the "copying assumption". If we know *something* of the amounts of branching, and the minimum probability of any branch, then we can infer *some things* about the number of test trials required to produce a critical mass of evidence for or against an hypothesis. In general, this type of information may be quite vague, but enough to give us a rough guesstimate on the number of test trials required.

The second issue concerns the test statistic itself. Take as an example the χ^2 statistic that we have been using. The power of χ^2 to detect an underlying disagreement between the theory and and sample data is largely controlled by the size of the sample. χ^2 is a continuous distribution but we have applied it to discrete data. This incurs approximations. With large expected frequencies this approximation is good, however this is not so with small frequencies. [Coc52] examines this subject in detail, and [DH70] discusses the use of Yates's correction when dealing with small expected frequencies, but it is customary to recommend that the smallest expected number should be between 5 and 10. Applying this rule-of-thumb to the smallest expected frequency in the table above, i.e. to expected frequency for $= bc \Rightarrow$, gives $\frac{9}{30}N \geq 10$, therefore N ought to be at least 34.

The reader may be concerned that 34 is markedly less than the figure 726 calculated earlier. The figure 34 has been calculated for a different reason to the figure 726. 34 is a heuristic value for the number of test trials that ought to be conducted in order to ensure that the discrete, sample distribution is a reasonable approximation to the continuous, χ^2 distribution. Whereas, 726 is the number of test trials that would have to be conducted before obtaining a critical mass of evidence to justify rejecting the H_0 hypothesis. The figure 726 may seem high, but remember that we formulated the hypotheses to significantly bias H_0, consequently we require a large amount of evidence against H_0 before justifying its rejection.

Sub-hypothesis 3

$H_{3.0}$: $imp \models \prec\prec b, d \succ, \frac{1}{3} \succ$
$H_{3.1}$: $imp \not\models \prec\prec b, d \succ, \frac{1}{3} \succ$

outcome	expected	observed
$= bd \Rightarrow$	N	N
$\neq bd \Rightarrow$	N	N
	N	N

$$\chi^2_{obs} = \sum \frac{(O-E)^2}{E} = \frac{(\frac{1}{3}N - \frac{1}{3}N)^2}{\frac{1}{3}N} + \frac{(\frac{2}{3}N - \frac{2}{3}N)^2}{\frac{2}{3}N} = 0$$

Therefore $\neg(\chi^2_{obs} > 3.84) \Rightarrow$ accept H_0
Again, test observations for the implementation in figure 7.34 would lead us to make this decision for any number N of test trials ≥ 1.

7.5.8 Discussion

We have considered PbLOTOS specifications to be mathematical objects, and their final realizations to be real world objects. The question 'is object Q a valid refinement of object P according to some probabilistic implementation relation' is posed as a hypothesis. Then we say that we can have total confidence in our hypothesis decision if both objects Q and P are PbLOTOS specifications (i.e. objects in the mathematical world). However, if at least one of these objects is a real world implementation, then we can have only some level of statistical confidence in our hypothesis decision.

- mathematical objects $\Rightarrow$ total confidence in hypothesis decision
- real world objects $\Rightarrow$ statistical confidence in hypothesis decision

Our approach to implementation relations for probabilistic systems has been much more pragmatic than approaches taken by [LS89, BM89, BIM88, vGSST90, GJS90, Han90]. They do not always make a distinction between mathematical specifications and real world implementations. [LS89] use "copying" [Abr87] and "minimum probability assumptions" [HM80] in their methods. However we have taken the view that such assumptions are unrealistic and too simplistic when considering real world probabilistic implementations, and instead statistical testing ought to be used. Moreover, when considering probabilistic specifications (not real world implementations), we believe that there is no need to use assumptions such as "copying" and "minimum probability" in a method for establishing validity, since specifications are mathematical objects whose details are completely knowable.

7.6 An example: a microprocessor CIM cell

This section shows how PbLOTOS can be used in the specification and testing of a simple CIM (Computer Integrated Manufacturing) system which manufactures microprocessors. A microprocessor CIM cell is specified using LOTOS. Then a refinement of this specification is written in PbLOTOS. The PbLOTOS specification formally describes an important probabilistic characteristic of the manufacturing cell. We show that the PbLOTOS specification is a valid refinement of the LOTOS specification according to the *probabilization* relation defined in section 7.4.11.1. We discuss how the testing framework developed in section 7.5 provides a basis for statistically testing a real world implementation of the manufacturing cell against its PbLOTOS specification.

7.6.1 The LOTOS specification

The LOTOS specification in figure 7.35 describes how the microprocessor CIM cell takes a *raw_silicon_wafer* and produces a *full_speed_processor*, a *reduced_speed_processor* or a *reject_processor*. (This scenario is typical of microprocessor manufacture. Some of the manufactured microprocessors will operate at the target clock speed. Others will fail to operate at the target clock speed but will operate at a reduced clock speed. Others will be totally rejected.)

```
process processor_CIM_cell1[raw_silicon_wafer,full_speed_processor,
                            reduced_speed_processor,reject_processor] :noexit :=
   raw_silicon_wafer;
   (
      i;
      full_speed_processor;
      processor_CIM_cell1[raw_silicon_wafer,full_speed_processor,
                          reduced_speed_processor,reject_processor]
   []
      i;
      reduced_speed_processor;
      processor_CIM_cell1[raw_silicon_wafer,full_speed_processor,
                          reduced_speed_processor,reject_processor]
   []
      i;
      reject_processor;
      processor_CIM_cell1[raw_silicon_wafer,full_speed_processor,
                          reduced_speed_processor,reject_processor]
   )
endproc (* processor_CIM_cell1 *)
```

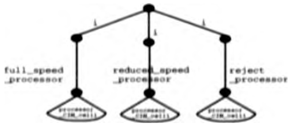


Figure 7.35: LOTOS specification of processor_CIM_cell1

7.6.2 The PbLOTOS specification

The LOTOS specification of the manufacturing cell does not contain any requirements about the relative probabilities of manufacturing *full_speed_processors*, *reduced_speed_processors* and *reject_processors*. Normally the specifier will want include requirements in the specification that tell the implementer to build a manufacturing cell which manufactures a high proportion of *full_speed_processors* and a low proportion of *reject_processors*. It is not possible to specify probabilistic behaviour in LOTOS, but it is possible in PbLOTOS. The PbLOTOS specification in figure 7.36 contains requirements for the probabilistic behaviour of the manufacturing cell. The specification requires that the manufacturing cell produces *full_speed_processors*, *reduced_speed_processors* and *reject_processors* in the ratio 6:3:1.

```
process processor_CIM_cell2[raw_silicon_wafer,full_speed_processor,
                            reduced_speed_processor,reject_processor] :noexit :=
   raw_silicon_wafer;
   (
       full_speed_processor;
       processor_CIM_cell2[raw_silicon_wafer,full_speed_processor,
                           reduced_speed_processor,reject_processor]
   [=0.6]
      (
          reduced_speed_processor;
          processor_CIM_cell2[raw_silicon_wafer,full_speed_processor,
                              reduced_speed_processor,reject_processor]
      [=0.75]
          reject_processor;
          processor_CIM_cell2[raw_silicon_wafer,full_speed_processor,
                              reduced_speed_processor,reject_processor]
      )
endproc (* processor_CIM_cell2 *)
```

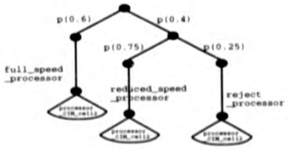


Figure 7.36: PbLOTOS specification of processor_CIM_cell2

In real life we may want to specify that of the microprocessors manufactured by the cell, *at least* 60% ought to be *full_speed_processors*, and of the percentage that are not *full_speed_processors*, *at least* 75% of these ought to be *reduced_speed_processors*. In order words, it might be useful to define additional operators for PbLOTOS such as $[< \mu]$, $[\leq \mu]$, $[\geq \mu]$, $[> \mu]$. Then we could write the specification:

```
process processor_CIM_cell3[raw_silicon_wafer,full_speed_processor,
                                        reduced_speed_processor,reject_processor] :noexit :=
   raw_silicon_wafer
   (
        full_speed_processor;
        processor_CIM_cell3[raw_silicon_wafer,full_speed_processor,
                               reduced_speed_processor,reject_processor]
   [≥ 0.6]
      (
         reduced_speed_processor;
         processor_CIM_cell3[raw_silicon_wafer,full_speed_processor,
                                reduced_speed_processor,reject_processor]
      [≥ 0.75]
         reject_processor;
         processor_CIM_cell3[raw_silicon_wafer,full_speed_processor,
                                reduced_speed_processor,reject_processor]
      )
endproc (* processor_CIM_cell3 *)
```

We identify defining additional PbLOTOS operators, as future work in section 9.3.

7.6.3 Proving the validity of a specification

Is the PbLOTOS specification of the microprocessor CIM cell a valid refinement of the LOTOS specification? To answer this question we need a definition of the phrase *valid refinement*. The *probabilization* relation ($\underline{prob}$) defined in section 7.4.11.1 provides a definition of what it means for one specification to be a valid refinement of another specification.[12] The *probabilization* relation checks if trace and probabilistic refusal properties are preserved in a refinement (see section 7.4.11.3 for examples).

So now we can pose the question formally as:

$processor\_CIM\_cell2 \;\; \underline{prob} \;\; processor\_CIM\_cell1$?

In section 7.4.11.1 the $\underline{prob}$ relation is defined in terms of the *SimChar* algorithm. Applying *SimChar* to the *cell1*[13] specification we get:

- the *set of trace probabilities* of *cell1* to be:

$$\{ \quad \prec\prec silicon, full\_speed \succ, \mu_1 \succ, \\ \prec\prec silicon, reduced\_speed \succ, \mu_2 \succ, \\ \prec\prec silicon, reject \succ, \mu_3 \succ \quad \}$$

- the *set of auxiliary equations* of *cell1* to be:

$$\{ \quad 0 < \mu_1 < 1, \\ 0 < \mu_2 < 1, \\ 0 < \mu_3 < 1, \\ \mu_1 + \mu_2 + \mu_3 = 1 \quad \}$$

[12] $\underline{prob}$ is defined for PbLOTOS specifications. Since LOTOS is a subset of PbLOTOS, we can treat the LOTOS specification *processor_CIM_cell1* as a PbLOTOS specification.

[13] For convenience we will use shortened forms of some of the specification identifiers.

Applying *SimChar* to the *cell2* specification we get:

- the *set of trace probabilities* of *cell2* to be:

$$\{ \prec\prec silicon, full\_speed \succ, 0.6 \succ, \prec\prec silicon, reduced\_speed \succ, (0.4 \times 0.75) \succ, \prec\prec silicon, reject \succ, \mu_3(0.4 \times 0.25) \succ \}$$

- the *set of auxiliary equations* of *cell2* to be:

$$\{\}$$

Now, *cell2* $\underline{prob}$ *cell1* *iff* (section 7.4.11.1):

(i) the trace sets of *cell2* and *cell1* are equal, and

(ii) it is always possible to find solutions to the *auxiliary equations* of *cell1* such that the probabilities of *cell1* and *cell2* traces are identical, for all possible solutions to the *auxiliary equations* of *cell2*.

It is trival to see that the trace sets of *cell2* and *cell1* are equal. Also, it is trival to see that we can find values for μ_1, μ_2 and μ_3 of *cell1* such that the probabilities of *cell1* and *cell2* traces are identical. Therefore *cell2* $\underline{prob}$ *cell1*, i.e. the PbLOTOS specification of the microprocessor CIM cell a valid refinement of the LOTOS specification.

(See appendix H for a more detailed example of the application of *SimChar*, and see section 7.4.11.3 for more complex examples of the $\underline{prob}$ relation.)

7.6.4 Testing a real world implementation

Section 7.6.3 showed how to prove that a probabilistic specification implemented (refined) another specification. In this section we look at how to test if a real world implementation correctly implements a probabilistic specification. We use the framework for testing described in section 7.5.

For this example, we assume that the testing is carried out by the quality control department at the company that manufactures microprocessor CIM cells. The company constructs the testing hypotheses with the base belief, H_0, that their product satifies the specification. The means that, unless testing provides a significant amount of evidence to the contrary, the company assume that their product satifies the specification (i.e. innocent unless 'proven' guilty).[14] Hence the hypotheses are formulated as:

$H_0 : imp \models cell2$
$H_1 : imp \not\models cell2$

where *cell2* is an abbreviation for the PbLOTOS specification *processor_CIM_cell2*, and *imp* denotes a real world implementation of a microprocessor CIM cell.

[14]Section 7.5.5 discusses the factors which influence the ways in which testing hypotheses may be constructed.

The company want to reduce the chance of rejecting implementations when they are in fact correct (true H_0). In hypotheses testing terms, the company want to reduce the chances of making Type I errors and so they set low significance level (α), such that $P\{Type\ I\ error\} \leq 0.05$ (see section 7.5.7). Looking up the test statistic χ^2 in tables [Fel68] they find that $\chi^2_{0.05,1} = 3.84$. So the company rejects H_0 if $\chi^2_{obs} > 3.84$ (i.e. the company rejects an implementation if the evidence against the implementation being correct is above the chosen significance level).

The company are particularly interested in testing the probabilistic behaviour of microprocessor CIM cells. So they take $\models$ to mean <u>*prob–eq*</u> (defined in section 7.4.11.2). In section 7.6.3 the specification *cell2* was characterized by a set of probabilistic traces using the *SimChar* algorithm. We call these traces 'properties' of *cell2*. An implmentation satisfies the specification *cell2*, if it satisfies each of the properties of *cell2* (as explained in section 7.5.7.1). The company formulate each property as a sub-hypothesis, and test each of these sub-hypotheses separately.

The properties of *cell2* are the probabilistic traces of *cell2*. These were established in section 7.6.3, and are repeated below.

$$\{ \quad \prec\prec silicon, full\_speed \succ, 0.6 \succ, \\ \prec\prec silicon, reduced\_speed \succ, (0.4 \times 0.75) \succ, \\ \prec\prec silicon, reject \succ, \mu_3(0.4 \times 0.25) \succ \quad \}$$

The probabilistic traces are formulated as sub-hypotheses as follow:

$H_{1,0}$: $imp \models \prec\prec silicon, full\_speed \succ, 0.6 \succ$
$H_{1,1}$: $imp \not\models \prec\prec silicon, full\_speed \succ, 0.6 \succ$

$H_{2,0}$: $imp \models \prec\prec silicon, reduced\_speed \succ, 0.3 \succ$
$H_{2,1}$: $imp \not\models \prec\prec silicon, reduced\_speed \succ, 0.3 \succ$

$H_{3,0}$: $imp \models \prec\prec silicon, reject \succ, 0.1 \succ$
$H_{3,1}$: $imp \not\models \prec\prec silicon, reject \succ, 0.1 \succ$

The company statistically test each implementation of the microprocessor CIM cell against each of the sub hypotheses. Each test of an implemention against a sub-hypothesis would be documented as as shown for sub-hypothesis 1 below (also see section 7.5.7.3).

Sub-hypothesis 1

$H_{1,0}$: $imp \models \prec\prec silicon, full\_speed \succ, 0.6 \succ$
$H_{1,1}$: $imp \not\models \prec\prec silicon, full\_speed \succ, 0.6 \succ$

outcome	expected	observed
$= silicon, full\_speed \Rightarrow$	$0.6N$	$\mu_{obs}N$
$\neq silicon, full\_speed \Rightarrow$	$0.4N$	$(1-\mu_{obs})N$
	N	N

where N is the number of test trials, and μ_{obs} is the fraction of these trails which provide evidence for $H_{1.0}$.

$$\chi^2_{obs} = \sum \frac{(O-E)^2}{E} = \frac{(\mu_{obs}N - 0.6N)^2}{0.6N} + \frac{((1-\mu_{obs})N - 0.4N)^2}{0.4N}$$

If $\chi^2_{obs} > 3.84 \Rightarrow$ accept $H_{1.0}$.

The number of test trials N should be as large as possible (see the text under Sub-hypothesis 2 in section 7.5.7.3).

Each sub-hypothesis test for each implementation would be laid out as shown for sub-hypothesis 1.

7.6.5 Discussion

This section has demonstrated the use of the *prob* and *prob−eq* relations (defined in sections 7.4.11.1 and 7.4.11.2), the *SimChar* algorithm (defined in section s:probrel), and the framework for statistical testing (described in section 7.5). The *prob* relation was used to prove that one probabilistic specification was a valid refinement of another specification. A framework for statistical testing was established to test if a real world implementation was correct with respect to (*prob−eq* to) a probabilistic specification.

7.7 Summary

The primary concern of this chapter has been to extend the LOTOS language and support theory to support the specification and development of probabilistic systems.

We extended the definition of LTSs (Labelled Transition Systems) to define NP-LTSs (Non-deterministic and Probabilistic LTSs) and P-LTSs (Probabilistic LTSs). NP-LTSs are LTSs which may contain both non-deterministic and probabilistic transitions. P-LTSs are LTSs which contain only probabilistic transitions.

NP-LTSs were used as a semantic model for PbLOTOS (Probabilistic LOTOS). PbLOTOS is LOTOS enhanced by a small number of syntactic and semantic extensions which support probabilistic features. PbLOTOS has a probabilistic choice operator for specifying probability distributions over a set of internal probability transitions.

Having defined PbLOTOS, the chapter turned to defining implementation relations (pre-orders) to support the development of PbLOTOS systems. We defined the *probabilization* pre-order (*prob*) as a formal implementation relations between NP-LTSs. "Probabilization" [BM89] of an NP-LTS involves replacing non-deterministic transitions by probabilistic transitions. In this way, we can consider that an NP-LTS S describes a set of implementations $\{I | I$ *prob* $S\}$ (P-LTSs).

We showed how to characterize the set of probabilistic implementations $\{I | I$ *prob* $S\}$ as a set of simultaneous equations, where each equation describes the probability of an observable trace of S. The *free-terms* within the simultaneous equations generate the *set* of probabilistic implementations. These free-terms are used to range over the possible probabilizations of non-deterministic branches in S. We defined an algorithm *SimChar* which, given an NP-LTS, generates a characterising set of simultaneous equations.

We saw how for any two NP-LTSs, S_1 and S_2, S_1 $\underline{prob}$ S_2 *iff* S_1 describes only a subset of the probabilistic implementations which S_2 describes, i.e. $\{I | I \ \underline{prob}\ S_1\} \subseteq \{I | I \ \underline{prob}\ S_2\}$. Furthermore, S_1 and S_2 are *probabilization equivalent*, written S_1 $\underline{prob-eq}$ S_2, *iff* $\{I | I \ \underline{prob}\ S_1\} = \{I | I \ \underline{prob}\ S_2\}$.

In contrast to other probabilistic process algebras, we moved the emphasis away from the semantics of the PbLOTOS language, and instead placed many of the 'probabilistic concepts' in the associated theory of relations. The consequence of this is that the probabilistic aspects of the PbLOTOS semantics are simpler than the probabilistic aspects of the semantics of other process algebra, but PbLOTOS theory for relations is, consequently, more complex.

We suggested a simple statistical testing framework for establishing whether a probabilistic implementation (a P-LTS) is a valid implementation of a probabilistic specification (an NP-LTS), according to the probabilization relation. We investigated if the "copying" [Abr87] and "minimum probability" [HM85] assumptions could be used as a basis for interpreting test observations and making hypothesis decisions about probabilistic systems. However, we decided that these assumptions were unrealistic, and abandoned them in favour of an established general purpose statistical inference method (χ^2).

We considered PbLOTOS specifications to be mathematical objects, and their final realizations to be real world objects. Then we said that we can have total confidence in a hypothesis decision about the validity of a relation between two objects Q and P, if both objects Q and P are PbLOTOS specifications (i.e. objects in the mathematical world). However, if at least one of these objects is a real world implementation, then we can have only some level of statistical confidence in our hypothesis decision.

Finally, we illustrate the use of the $\underline{prob}$ relation, the $\underline{prob-eq}$ relation, the *SimChar* algorithm and the framework for statistical testing in an example of the development of a simple microprocessor CIM cell.

Chapter 8

The specification of priority, and Extended LOTOS

This chapter comes in two parts. The first part extends LOTOS with features for the specification of priority events. We call the result PrLOTOS. Priority semantics ensure that those events with the highest priority weighting of their class are fired first.

The second part of the chapter describes how quantitative timing extensions (TLOTOS) (chapter 6), probability extensions (PbLOTOS) (chapter 7) and priority extensions (PrLOTOS) (this chapter) integrate to form Extended LOTOS (XL). Also, we describe a simple example specification which illustrates the expressive flexibility of XL.

8.1 The formal specification of priority

This section defines PrLOTOS — LOTOS extended with features for the specification of priority. We describe the new priority features, provide simple examples of their use, and define the syntactic and semantic extensions to LOTOS for their support.

8.1.1 Introduction

The priority feature of PrLOTOS facilitates the specification of priority among events which are mutually exclusive alternatives in a choice. To specify priority information for an event, the syntax $\#(v_c, v_p)$ is included in the event denotation, where v_c (a *Nat* sort) indicates the ***priority-class*** of the event, and v_p (a *Nat* sort) indicates the ***priority-value*** (or priority weighting) of the event. The higher the ***priority-value***, the higher the priority of the event (within its ***priority-class***).

Where there is a choice between events, all of which belong to the same ***priority-class***, the event with the highest ***priority-value*** will be fired (or there will be a non-deterministic choice between those events with equal highest ***priority-value*** within the ***priority-class***). Where there is a choice between events, of more than one ***priority-class***, there will be a non-deterministic choice between the events which are highest in their respective ***priority-classes***. Each event with an unspecified priority (an unprioritized event) is deemed to be in a unique ***priority-class***, and hence will cause a non-deterministic choice as an alternative in a ***choice-expression***.

If two event offers synchronize then either at least one of them must be unprioritized, or they must both be of the same ***priority-class***.

If two prioritized event offers synchronize then they must both be of the same ***priority-class***. The resultant event offer will be of this ***priority-class*** and have a ***priority-value*** equal to the higher of the two ***priority-values*** of the synchronizing event offers.

If an unprioritized event offer synchronizes with a prioritized event offer, then the result is an event offer with the same ***priority-class*** and the same ***priority-value*** as the prioritized event offer. We say that the unprioritized event offer is ***associated*** with the prioritized event offer. In this way, priority information, localized in one process, can influence, through synchronization, the prioritization of events with unprioritized event offers in other processes. We call this the ***association*** feature.

An unprioritized event is given a 'status' equal to that of a prioritized event. A choice between an unprioritized event and a prioritized event is a non-deterministic choice. We consider that an unprioritized event is an event without a specified priority — not necessarily an event with a low priority.

Our general approach to priority specification is that no priority precedence is assumed unless explicitly specified. No precedence is assumed between ***priority-classes***, or between unprioritized events.

8.1.1.1 Related work

Extreme cases of van Glabbeek *et al.*'s ***stratified*** model (section 7.2.1), in which zero probabilities are permitted, can be used to describe a notion of priority. Consider the stratified model expression:

$$1a + 0(1b + 0c)$$

This gives a priority over b and c, and b priority over c. c can occur only in a restricted environment in which a and b cannot occur. b can only occur in a restricted environment in which a cannot occur.

van Glabbeek *et al.*'s priority model includes features that we wish PrLOTOS to include. However, we do not use the stratified probabilistic model approach as basis for PrLOTOS.

8.1.2 Syntax extensions

8.1.2.1 PrLOTOS action-denotations

The following ***special-symbol*** is added to [ISO89b, clause 6.1.3.2] (and removed from ***special-character*** clause 6.1.2 to prevent parsing problems):

```
priority-symbol =    "#".
```

and alter the ***action-denotation*** section 6.5.2 to:

```
action-denotation =   gate-identifier
                      [ experiment-offer { experiment-offer } [ selection-predicate ] ]
                      [ time-offer ] [ time-policy ] [ time-establishment ]
                      [ priority-symbol open-parenthesis-symbol priority-class
                      comma-symbol
                      priority-value-symbol close-parenthesis-symbol ]
                      | internal-event-symbol
                      [ time-offer ] [ time-policy ] [ time-establishment ]
                      [ priority-symbol open-parenthesis-symbol priority-class
                      comma-symbol
                      priority-value-symbol close-parenthesis-symbol ].
priority-class =      term-expression.
priority-value =      term-expression.
```

Constraint: The ***term-expressions*** within ***priority-class*** and ***priority-value*** must evaluate to the sort *Nat*.

Priority constraints may be associated with observable and internal events. Our semantic extensions for priority event ordering ensure that those events with the highest priority weighting (***priority-value***) of their ***priority-class*** are eligible to be fired first.

The evaluation order of terms within an ***action-denotation***, is as follows:

1. Firstly, the ***experiment-offers*** and ***selection-predicate*** are processed (as described in section 6.5.1.5).

2. Then, the ***time-offer*** and ***time-policy*** are processed (as described in section 6.5.1.5).

3. Finally, the ***priority-class*** and ***priority-value*** are processed.

8.1.2.2 PrLOTOS examples

u #(1,4); P []v #(1,8); Q

> This is an example of a choice between two observable events of the same *priority-class* with different *priority-values*.
>
> Event *v* is in the same *priority-class* as event *u* (class 1) and credited with a higher priority weighting (8 > 4). Therefore if events *u* and *v* are both firable, event *v* will be fired.

i #(1,4); P []**i** #(1,8); Q

> This is an example of a choice between two unobservable events of the same *priority-class* with different *priority-values*.
>
> The leftmost **i** event will never occur. The rightmost **i** event will pre-empt the leftmost **i** event since both events are in the same priority class and the rightmost **i** event has a higher priority weighting than the leftmost **i** event.

u #(1,4); P []u #(1,8); Q

> This is an example of a choice between two identical observable events of the same *priority-class* with different *priority-values*, at the same gate.
>
> The leftmost *u* event will never occur. The rightmost *u* event will pre-empt the leftmost *u* event.

i #(1,4); P []v #(1,8); Q

> This is an example of a choice between an observable event and an unobservable event of the same *priority-class* with different *priority-values*.
>
> The **i** event will only occur if the *v* event cannot be fired (fails to synchronize). This template could be used to specify a "fail safe" or "fall back" mechanism within a system, where the **i** event represents a fall-back action, to be taken only if *v* could not occur. In a real specification a "fall back" mechanism might also include quantitative timing constraints such as "the fall back action **i** is enabled only after some time *t*".

u #(1,4); P []v #(2,8); Q

> This is an example of a choice between two observable events of different *priority-classes*.
>
> No priority relationship exists between events *u* and *v* because they are not of the same *priority-class*. Hence, if both *u* and *v* events can occur, then there is a non-deterministic choice between these events.

u #(1,4); P []v #(1,4); Q

> This is an example of a choice between two observable events of the same *priority-class* with the same *priority-values*.
>
> Events *u* and *v* are in the same *priority-class* and have the same *priority-value*. Hence, if both *u* and *v* events can occur, then there is a non-deterministic choice between these events.

u #(1,4); P []v #(1,8); Q []w; R

> This is an example of a choice between three observable events, two of which are of the same *priority-class*.
>
> If all three events u, v and w are firable, then there will be a non-deterministic choice between only events v and w. Event u will be pre-empted by event v because both events are in the same *priority-class*, but event v has a higher *priority-value* than u.

(u #(1,4); P []v #(1,8); P) || Q[u,v]

> This is an example of the *association* feature.
>
> Assume that process Q contains no prioritized events. Then all priority information has been localized within the parenthesized behaviour expression (conforming with the 'separation of concerns' principle, section 4.2.2.3). The *association* feature of PrLOTOS allows the parenthesized behaviour expression to govern the prioritization of events in the process Q by synchronizing with these events.

(u #(1,4); P []v #(1,8); P) |[u]| (u #(1,9); Q []w #(1,6); Q)

> This is an example of priority negotiation between synchronizing *parallel-expressions*.
>
> The result of the parallel combination of the two *choice-expressions* is to prioritize u with a *priority-value* of 9 (the higher of the *priority-values* from the two u event offers). Hence, considering the PrLOTOS expression in isolation, event u may be fired since it has a higher *priority-value* (9) than either v (8) or w (6).

8.1.3 Semantic Extensions

8.1.3.1 Extensions to the structure of a transition

Section 6.5.4.1 defined the structure of a timed transition as $TT = -aTHt\rightarrow$. Here we extend the structure of a TT to: $TT = -aTHtv_cv_p\rightarrow$, where

- a, T, H and t are defined as described in section 6.5.4.1.
- v_c is a *priority-class*.
- v_p is a *priority-value*.

8.1.3.2 Additional transition rule schemas

The following transition rule schemas realize priority event ordering in PrLOTOS.

Only those schemas that are significant for the operation of priority are given, i.e. we describe only the schemas responsible for modifying *priority-class* or *priority-value* terms. These schemas (below) need to be integrated, in the obvious way, with the semantic schemas in sections refs:tlsem and 7.3.3. The remaining schemas, not given below, are those in section 6.5.4, with simple extensions to the transition structures as described above, to carry v_c and v_p terms.

Choice-expressions

if

> B, B_2 are *parallel-expressions*,
> B_1 is a *choice-expression*,
> B'_1, B'_2 are *behaviour-expression* instances,
> $a_1, a_2 \in Act$,
> $t, t', t_1, t'_1, t_2, t'_2$ are ground terms of sort *TimeSort*,
> T, T_1, T_2 are ground terms of sort *TimeSetSort*,
> H, H_1, H_2 are terms of sort *NegotiatedTimePolicySort*,
> v_{c1}, v_{c2} are *priority-class* terms,
> v_{p1}, v_{p2} are *priority-value* terms

with

$$\prec B, t \succ = \prec B_1 [] B_2, t \succ$$

then

$$\frac{\prec B_1, t \succ -a_1 T_1 H_1 t'_1 v_{c1} v_{p1} \rightarrow \prec B'_1, t'_1 \succ, \quad \prec B_2, t \succ -a_2 T_2 H_2 t'_2 v_{c2} v_{p2} \rightarrow \prec B'_2, t'_2 \succ}{\prec B, t \succ -a_1 T H t' v_{c1} v_{p1} \rightarrow \prec B'_1, t' \succ}$$

and $((v_{c1} = undefined) \wedge (v_{c2} = undefined)) \vee (v_{c1} \neq v_{c2}) \vee ((v_{c1} = v_{c2}) \wedge (v_{p1} \geq v_{p2}))$.

Note: We assume that a slightly extended version of the flattening function (section 6.5.3.2) assigns the special value *undefined* to *priority-class* and *priority-value* of an event with unspecified priority.

a_1 can be fired: (1) if the priority of either event is unspecified; or (2) if a_1 is not in the same *priority-class* as a_2; or (3) if a_1 is in the same *priority-class* as a_2 and its *priority-value* is greater than or equal to a_2's *priority-value*.

$$\frac{\prec B_1, t \succ -a_1 T_1 H_1 t'_1 v_{c1} v_{p1} \rightarrow \prec B'_1, t'_1 \succ, \quad \prec B_2, t \succ -a_2 T_2 H_2 t'_2 v_{c2} v_{p2} \rightarrow \prec B'_2, t'_2 \succ}{\prec B, t \succ -a_2 T H t' v_{c2} v_{p2} \rightarrow \prec B'_2, t' \succ}$$

and $((v_{c1} = undefined) \wedge (v_{c2} = undefined)) \vee (v_{c1} \neq v_{c2}) \vee ((v_{c1} = v_{c2}) \wedge (v_{p2} \geq v_{p1}))$.

Note: a_2 can be fired: (1) if the priority of either event is unspecified; or (2) if a_2 is not in the same *priority-class* as a_1; or (3) if a_2 is in the same *priority-class* as a_1 and its *priority-value* is greater than or equal to a_1's *priority-value*.

Parallel-expressions

if

B, B_2 are *parallel-expressions*,
B_1 is a *choice-expression*,
B'_1, B'_2 are *behaviour-expression* instances,
$a, a' \in Act$,
$t, t', t_1, t'_1, t_2, t'_2$ are ground terms of sort $TimeSort$,
T, T_1, T_2 are ground terms of sort $TimeSetSort$,
H, H_1, H_2 are terms of sort $NegotiatedTimePolicySort$,
v_{c1}, v_{c2} are *priority-class* terms,
v_{p1}, v_{p2} are *priority-value* terms,
$g_1, \ldots, g_n$ is a (possibly empty) list of *gate-name* instances

with

$\prec B, t \succ = \prec B_1 |[g_1, \ldots, d_n]| B_2, t \succ$

then

$$\frac{\prec B_1, t \succ \; -aT_1 H_1 t'_1 v_{c1} v_{p1} \rightarrow \prec B'_1, t'_1 \succ, \quad \prec B_2, t \succ \; -aT_2 H_2 t'_2 v_{c2} v_{p2} \rightarrow \prec B'_2, t'_2 \succ}{\prec B, t \succ \; -aTHt'v_c v_p \rightarrow \prec B'_1 |[g_1, \ldots, d_n]| B'_2, t' \succ}$$

and $name(a) \in \{g_1, \ldots, g_n, \delta\}$,
$CanSynchronise(v_{c1}, v_{c2}) = True$,
$v_c = NegotiateClass(v_{c1}, v_{c2})$,
$v_p = NegotiatePrValue(v_{c1}, v_{c2}, v_{p1}, v_{p2})$

where

```
CanSynchronise($v_{c1}, v_{c2}$) =
      if ($v_{c1} = v_{c2}$) then True
      elseif (($v_{c1}$ = undefined) or ($v_{c2}$ = undefined)) then True
      else False
      endif.

NegotiateClass($v_{c1}, v_{c2}$) =
      if ($v_{c1} = v_{c2}$) then $v_{c1}$
      elseif ($v_{c1}$ = undefined) then $v_{c2}$
      elseif ($v_{c2}$ = undefined) then $v_{c1}$
      else undefined
      endif.
```

```
$NegotiatePrValue(v_{c1}, v_{c2}, v_{p1}, v_{p2})$ =
    if (($v_{c1} = v_{c2}$) or ($v_{c1}$ = *undefined*) or ($v_{c2}$ = *undefined*)) then
        if ($v_{p1} \geq v_{p2}$) then $v_{p1}$
        else $v_{p21}$
        endif
    else *undefined*
    endif
```

Note: The semantics for priority place a further restriction on process synchronization: for two event offers to synchronize they must be of the same *priority-class*, or at least one of them must be unprioritized.

If synchronization occurs and both event offers are unprioritized, then the resultant event offer is unprioritized.

If synchronization occurs and only one of the event offers is unprioritized, then the resultant event offer takes the *priority-class* and *priority-value* of the prioritized event offer. This rule realizes the *association* feature that we introduced in section 8.1.1.

If synchronization occurs and both event offers are prioritized, then they must both be of the same *priority-class*. The resultant event offer will be of this *priority-class* and have a *priority-value* equal to the higher of the two values v_{p1} and v_{p2}.

If the two event offers are prioritized but have different *priority-classes*, then synchronization cannot occur.

8.1.4 Discussion

8.1.4.1 Choice as a result of interleaving

Prioritized event occurrence is enforced when choices occur as a direct result of a *choice-expression*. However, prioritized event occurrence is not enforced when choices occur as a direct result of the interleaving model of *parallel-expressions*. Enforcing prioritization of events when merging interleaved behaviours would be akin to "process scheduling". It would not be technically difficult to extend PrLOTOS semantics with a "process scheduler" with specifier-selectable "scheduling policies" for merging interleaved behaviours (*parallel-expressions*). However this work is outside the scope of this thesis. Nonetheless, PrLOTOS users can specify their own "process scheduler" at the syntax level. They can use the given priority features to build a "process scheduler", and then synchronize it with the *parallel-expression* in order to prioritize events within the *parallel-expression* (and hence govern the merging/interleaving of the parallel behaviours).

8.1.4.2 The *association* feature for separation of concerns

The "process scheduler" would be an example which makes use of the *association* feature of PrLOTOS. The necessary priority information can be confined to the "process scheduler". The "process scheduler" influences the prioritization of events offers outside itself by synchronization.

Isolating priority information in this way neatly separates priority requirements from other requirements. Examples which uses the *association* feature of PrLOTOS to separate priority concerns from other functional and performance concerns are: the penultimate expression in section 8.1.2.2, the *Precedence* process in the vending machine specification in section 8.2.3, the *Scheduler* example in section 8.1.5 and the *PRIORITY_SELECTION* process described in section 5.4.2.4.

8.1.4.3 Negotiating priority information for synchronizing events

In the semantics for *parallel-expressions* we define an algorithm which, in effect, negotiates the resultant *priority-class* and *priority-value* from the synchronization of two event offers. We can think of other algorithms for negotiating priority information.

For example, an alternative negotiation algorithm might realize the following negotiation policy[1]: when two event offers from the same *priority-class* synchronize, the resultant *priority-value* is a mean of the individual event offer *priority-values*.

PrLOTOS could be extended to allow users to express precedence among *priority-classes*. Another possible extension might allow prioritized events to belong to the *priority-class* '*any*'. When such an event offer with *priority-class any* synchronizes with an event offer with the (particular) *priority-class* v_c, then the *priority-value* of the resultant event would be a function of the two offered *priority-values*. The basis for deciding the *priority-class* of the resultant event would not be so clear. The resultant *priority-class* could be either *any* or v_c.

8.1.5 An example: specifying a job scheduler

In this subsection we show how PrLOTOS can be used to specify a simple job scheduler. Consider the following PrLOTOS specification:

```
process Scheduler [begin_crucial_job, end_crucial_job,
                   begin_non_crucial_job, end_non_crucial_job] :noexit :=

    begin_crucial_job #(1,2); (* highest priority *)
    end_crucial_job;
    Scheduler [begin_crucial_job,end_crucial_job,
               begin_non_crucial_job,end_non_crucial_job]
  []
    begin_non_crucial_job #(1,1); (* lowest priority *)
    end_non_crucial_job;
    Scheduler [begin_crucial_job,end_crucial_job,
               begin_non_crucial_job,end_non_crucial_job]

endproc (* Scheduler *)
```

[1] Similarities exist between the idea of negotiating *priority information* and the idea of negotiating *time-policy information* (section 6.5.4.1). Again, this idea generalizes to the idea of *set negotiation and narrowing* (section 6.5.4.4).

Note the following points about the *Scheduler* specification:

- A current job (i.e. a job being processed) is delimited by the events ***begin_crucial_job*** and ***end_crucial_job***, or by the events ***begin_non_crucial_job*** and ***end_non_crucial_job***.

- If a crucial job is available for processing at the same instant as a non-crucial job, the crucial job will be processed first. Crucial and non-crucial jobs are both in ***priority-class*** 1. Crucial jobs have a higher priority (***priority-value*** 2) than that of non-crucial jobs (***priority-value*** 1).

- In an environment (e.g. the CIM-OSA Business Complex, section 5.1.1.1) which continues to offer jobs until they are processed, non-crucial jobs are effectively queued awaiting the processing of all offered crucial jobs.

- The *Scheduler* process makes use of the ***association*** feature of PrLOTOS. It influences the prioritization of events offers outside itself by synchronization. The *Scheduler* process could be placed in a context where it synchronizes with job offers from other processes in order to prioritize these job offers. The ***association*** feature of PrLOTOS has allowed all of the job scheduling information to be confined the the *Scheduler* process.

It would be tedious to write a LOTOS specification with the same observable behaviour as the PrLOTOS specification. If we used LOTOS, we would have to build an explicit mechanism for queuing and dequeuing offered jobs. All offered jobs would have to be placed in this queue pending processing. A dequeuing function would have to remove crucial jobs from the queue before non-crucial jobs. Specifying the *Scheduler* in LOTOS would be tedious, but it would have a more serious drawback. The introduction of a queuing mechanism into the specification might be considered over-specification. It might be said that the description of a queuing mechanism introduces implementation bias.

This example has illustrated how PrLOTOS allows priority concerns to be easily expressed. The example has also shown how the ***association*** feature of PrLOTOS supports the separation of concerns.

8.1.6 Summary

This section has extended the LOTOS syntax and semantics with features for prioritizing events which form the alternatives in ***choice-expressions***.

A prioritized event is given a ***priority-class*** and ***priority-value***. Where there is a choice between events from the same ***priority-class***, the event with the highest ***priority-value*** will be fired. A choice between events from different ***priority-classes*** is rationalized to a non-deterministic choice between the events with the highest ***priority-value*** in their respective ***priority-class***. A choice between unprioritized events and prioritized events gives rise to non-deterministic choice. Prioritized event offers may synchronize with unprioritized event offers, thus prioritizing these unprioritized event offers through what we have called ***association***.

8.2 Extended LOTOS (XL)

This section describes XL as the integration of TLOTOS, PbLOTOS and PrLOTOS. Also we demonstrate the special expressiveness afforded by XL to performance concerns.

8.2.1 Introduction

Extended LOTOS (XL) is a formal specification language based on LOTOS. In addition to the features that LOTOS supports, XL also supports features for the formal specification of quantitative timing, probabilistic and priority concerns.

XL is formed by integrating the individual extensions to LOTOS: TLOTOS (chapter 6), PbLOTOS (chapter 7) and PrLOTOS (section 8.1).

8.2.2 TLOTOS+PbLOTOS+PrLOTOS=XL

TLOTOS, PbLOTOS and PrLOTOS may be used in isolation or in combination with one another. We call the combination of all three Extended LOTOS (XL).

Sections 6.5.1, 7.3.3 and 8.1.2 progressively define the syntactic extensions to LOTOS required to support XL. While, sections 6.5.2, 6.5.3, 6.5.4, 7.3.3 and 8.1.3 progressively define the semantic extensions to LOTOS required to support XL.

When integrated as XL, the interference between the TLOTOS, PbLOTOS and PrLOTOS parts is minimal (and well defined). The PbLOTOS part of XL does not directly interfere with interpretations of TLOTOS or PrLOTOS parts. This is because PbLOTOS is essentially concerned with only the *p-choice* operator (see section 7.3.3) which does not figure in the definitions of TLOTOS or PrLOTOS. Interference between the TLOTOS and PrLOTOS parts does not occur. PbLOTOS deals with the resolution of choice between possible events, arising from *choice-expressions*. TLOTOS deals with the resolution of choice between possible events, arising as a consequence of interleaving of *parallel-expressions*. TLOTOS must resolve choices between the interleaving order of events from *parallel-expressions* to ensure 'sensible interleaving' (see section 6.3.8). PrLOTOS must resolve choices between mutually exclusive events within a *choice-expression* in accordance with the priorities assigned to these events (see section 8.1.3).

8.2.3 An XL example

This subsection demonstrates the expressive flexibility of XL for the specification of performance concerns. To do this we use an example — the specification of a simple vending machine. We show how the informal requirements for the vending machine, especially performance related requirements, can be precisely and concisely captured in a XL specification. We further explain the XL specification by means of synchronizing finite state machines. Then we list the properties that we might wish a specification to possess, and relate these properties to the XL vending machine example.

Also see: sections 6.5.1 and 6.7 for examples specific to the specification of quantitative timing concerns; sections 7.3.4 and 7.6 for examples specific to probabilistic concerns;

sections 8.1.2 and 8.1.5 for examples specific to priority concerns; and sections 5.3.6 and 5.4.1 for two large examples which involve the specification of quantitative timing, probabilistic, priority, resource and functionality concerns.

8.2.3.1 The vending machine

The informal requirements for the vending machine are as follows.

- The function of the vending machine is to accept a coin and then issue a chocolate bar.
- The vending machine is always willing to repeat this sequence.
- However, the vending machine is a little unreliable: 2% of the time it will simply not issue a chocolate bar.
- To ensure that a hungry user does not have to wait too long for a chocolate bar, the time interval between the vending machine accepting a coin and issuing a chocolate bar is to be not more than 3 seconds.
- This vending machine has two coin slots: a £1 coin slot and a 50p coin slot. Only one of these coins is needed to pay for a bar of chocolate. However, if the user offers both types of coin (by placing a £1 coin in one slot and a 50p coin in the other slot), the machine will greedily choose to accept the £1 coin as payment, giving no change.

The following XL specification formalizes these informal requirements.

8.2.3.2 The XL specification

```
(* XL specification of vending machine *)
specification VendingMachine[coin_1pound,coin_50p,choc_bar] : noexit

  behaviour

    hide no_choc_bar in
    (
      Functionality[coin_1pound,coin_50p,choc_bar,no_choc_bar]
      ||
      (
        Reliability[choc_bar,no_choc_bar]
        |[choc_bar,no_choc_bar]|
        (
          QuantitativeTiming[coin_1pound,coin_50p,choc_bar,no_choc_bar]
           |[coin_1pound,coin_50p]|
           Precedence[coin_1pound,coin_50p]
        )
      )
    )

  where

    (* Functionality constraints... *)
    process Functionality[coin_1pound,coin_50p,choc_bar,no_choc_bar] : noexit :=
          (coin_1pound; exit [] coin_50p; exit) (* ...accept a coin *)
      >>
```

```
        (
          choc_bar; (* ...issue chocolate bar *)
          Functionality[coin_1pound,coin_50p,choc_bar,no_choc_bar] (* ...repeat *)
        []
          no_choc_bar; (* ...do not issue chocolate bar *)
          Functionality[coin_1pound,coin_50p,choc_bar,no_choc_bar] (* ...repeat *)
        )
    endproc (* Functionality *)

    (* Probabilistic constraints... *)
    process Reliability[choc_bar,no_choc_bar] : noexit :=
        (* 98% probability of chocolate bar *)
        choc_bar;
        Reliability[choc_bar,no_choc_bar]
      [=0.98]
        (* 2% probability of no chocolate bar *)
        no_choc_bar;
        Reliability[choc_bar,no_choc_bar]
    endproc (* Reliability *)

    (* Quantitative timing constraints... *)
    process QuantitativeTiming[coin_1pound,coin_50p,choc_bar,no_choc_bar] : noexit :=
          (coin_1pound @t1 ASAP; exit(t1) [] coin_50p @t1 ASAP; exit(t1))
                              (* ...accept a coin without delay, recording
                                   the time at which a coin is accepted *)
      >> accept t1:TimeSort in
         (
           i {setInterval(t1,t1+3)};
                                (* ...make chocolate bar ready within 3 seconds *)
           choc_bar ASAP; (* ...and immediately offer a chocolate bar *)
           QuantitativeTiming[coin_1pound,coin_50p,choc_bar,no_choc_bar]
         []
           no_choc_bar {setEQ(t1)}; (* ...no chocolate bar issued *)
           QuantitativeTiming[coin_1pound,coin_50p,choc_bar,no_choc_bar]
         )
    endproc (* QuantitativeTiming *)

    (* Priority constraints... *)
    process Precedence[coin_1pound,coin_50p] : noexit :=
      (* if both types of coin are offered then accept the 1pound_coin... *)
        coin_1pound #(1,2); (* higher priority (2) *)
        Precedence[coin_1pound,coin_50p]
      []
        coin_50p #(1,1); (* lower priority (1) *)
        Precedence[coin_1pound,coin_50p]
    endproc (* Precedence *)

endspec (* VendingMachine *)
```

We make a few comments about this specification:

- Notice how XL constructs support the separation of functionality, probabilistic, timing and priority concerns. We could have used a monolithic structure. This would have resulted in shorter but less understandable specification text.
- Notice the directness of expression which XL affords probabilistic, timing and priority concerns.
- The *Functionality* process captures the functionality requirements of the vending machine using standard LOTOS language constructs.

- The *Reliability* process captures the probabilistic requirements of the vending machine using the PbLOTOS language construct [= 0.98] (specifying the probability of issuing a *choc_bar*). (The unobservable *no_choc_bar* event is used to keep the specification processes in step when no chocolate bar is issued.)
- The *QuantitativeTiming* process captures the quantitative timing requirements of the vending machine using a number of TLOTOS language constructs. @*t1* is used to establish and record the time at which a coin is accepted. {*setInterval*(*t1*, *t1* + 3)} is used to specify the time at which a chocolate bar ought to be made available to the environment. **ASAP** is used to specify that the vending machine is willing to participate in observable events (i.e. interact with the user) as soon as possible (without introducing any unnecessary delay).
- The *Precedence* process captures the priority requirements of the vending machine using the PrLOTOS language constructs #(1,2) and #(1,1) (specifying the priorities assigned to *coin_1pound* and *coin_50p*, respectively).

Sections 5.3.6 and 5.4.1 discuss more detailed XL examples. The following subsection explains details of the XL specification, using synchronizing finite state machines.

8.2.3.3 An explanation using synchronizing finite state machines

Figure 8.1 explains the XL specification of the vending machine using synchronizing finite state machines (FSMs). The synchronizing FSMs representation is structured to reflect the parenthesised *parallel-expression* structure of the XL specification — the boxes in the figure represent the parentheses in the specification.

The FSMs synchronize on the transitions *50p*, *£1*, *choc* and *no_choc* which are synonyms for the XL events *coin_50p*, *coin_1pound*, *choc_bar* and *no_choc_bar*. In one sense the *tick* transitions are observable since we assume that the environment has the same notion of time-progress as the system itself.

- In the *Reliability* FSM, the probability transitions *p*(0.98) and *p*(0.02) denote the probabilities of taking the two possible paths 'issue chocolate bar' and 'do not issue chocolate bar'.
- In the *QuantitativeTiming* FSM, a *tick* transition represents the passing of 1 second of time. An *i(ready)* transition indicates the moment after which the vending machine makes a chocolate bar available. The FSM makes sure that, if the vending machine offers a chocolate bar, then the chocolate bar will be made available within 3 seconds of the vending machine accepting a coin. The *choc* transition occurs when the user actually accepts the chocolate bar from the machine. Notice that, in the FSM representation, we have not indicated the ASAP urgency specified for the *50p*, *£1* and *choc* transitions. This urgency exists in the XL specification, but reflecting it in the FSM representation would require the presentation of a lot of detailed 'machinery'.
- In the *Precedence* FSM, we use the three outlined arrows and states to reflect the XL mechanism responsible for establishing what coin combination is being offered to the vending machine.

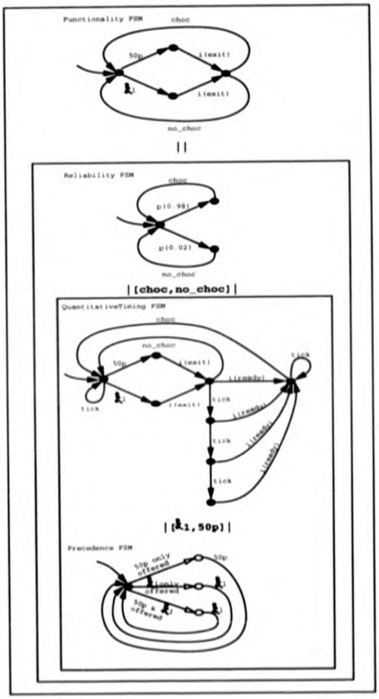


Figure 8.1: The XL specification explained using synchronizing FSMs

8.2.3.4 Properties of the XL specification

In this subsection we list properties that we might wish a specification to possess, and provide examples of these properties using the XL vending machine specification. (For the sake of readability, our descriptions of these properties are not strictly formal. The $= \sigma \Rightarrow$ notation for event sequences was introduced in section 7.3.5.)

Eventuality properties: properties that eventually will become true. Consider the following trivial example from our vending machine specification:

$$\mathit{VendingMachine} \models \text{eventually } \mathit{coin\_1pound}$$

i.e. the vending machine satisfies the property: the vending machine will eventually accept a £1 coin.

Reliability properties: properties that are true with a specified probability. For example:

$$\mathit{VendingMachine} \models P(=\mathit{choc\_bar}\Rightarrow \mid =\mathit{coin\_1pound}\Rightarrow) = 0.98$$

i.e. the vending machine satisfies the property: the probability of getting a chocolate bar given the acceptance of a £1 coin, is 0.98.

Invariance/Recurrence properties: properties that are always true, or are true infinitely often. For example:

$$\mathit{VendingMachine} \models \text{infinitely often } =\mathit{coin\_1pound}\Rightarrow$$

i.e. the vending machine satisfies the property: a £1 coin will be accepted infinitely often.

Precedence properties: properties that specify the ordering of events. For example:

$$\begin{aligned}\mathit{VendingMachine} \models\ & \text{if both } =\mathit{coin\_1pound}\Rightarrow\ \&\ =\mathit{coin\_50p}\Rightarrow \text{ are offered by the user}\\ & \text{then } =\mathit{coin\_1pound}\Rightarrow \text{ will occur}\end{aligned}$$

i.e. the vending machine satisfies the property: if both a £1 and a 50p coin are offered to the vending machine, the vending machine will accept the £1 coin and not accept the 50p coin.

Real-time properties: properties that will become true within some specified time. For example:

$$\begin{aligned}\mathit{VendingMachine} \models\ & =\mathit{coin\_1pound.choc\_bar}\Rightarrow \text{ implies}\\ & \text{clocktime}(=\mathit{choc\_bar}\Rightarrow) - \text{clocktime}(=\mathit{coin\_1pound}\Rightarrow) \leq 3\end{aligned}$$

i.e. the vending machine satisfies the property: if a chocolate bar is dispensed, it is dispensed at a time not greater than 3 seconds after a coin was accepted.

Performance properties: properties that with a specified probability become true within some specified time. For example:

$$\begin{aligned}\mathit{VendingMachine} \models P(&\text{clocktime}(=\mathit{choc\_bar}\Rightarrow) \leq t_1 + 3\\ &\mid \text{clocktime}(=\mathit{coin\_1pound}\Rightarrow) = t_1) = 0.98\end{aligned}$$

i.e. the vending machine satisfies the property: the probability of dispensing a chocolate bar within 3 seconds of accepting a £1 coin, is equal to 0.98.

8.2.4 Summary

This section described Extended LOTOS (XL) as the integration of TLOTOS (chapter 6), PbLOTOS (chapter 7) and PrLOTOS (section 8.1).

We discussed how the interference between the TLOTOS, PbLOTOS and PrLOTOS parts of XL is minimal. This allows any one of these three parts to be studied or used in isolation (as we have done in this thesis). Then we presented a simple example XL specification. The example illustrated how XL supports the syntactic and semantic separation of functionality, quantitative timing, probabilistic and priority concerns in the specification of a system in which these concerns interwork. Also, the example showed the directness of expression afforded by XL to quantitative timing, probabilistic and priority concerns. (A large example of XL applied to a CIM-OSA system is given in appendix B and explained in section 5.4.)

Chapter 9

Conclusions

This chapter summarizes the accomplishments of the work reported in this thesis and indicates possible future work.

9.1 General conclusion

This thesis constitutes a step in the evolution of distributed system specification methods that have both an architectural basis and a formal basis.

9.2 Overall summary of work

This section summarizes what we have accomplished in the thesis.

9.2.1 Key research issues for the thesis

In chapters 1, 2 and 3 we overviewed the topics of distributed computing systems and formal specification languages. From these topics we selected (sections 2.4 and 3.5) a number of key research issues for the thesis. The following subsections précis these issues and their research importance.

9.2.1.1 Architecture-driven specification

The nature of distributed systems makes them complex to specify and design (section 2.2). We have promoted the notion of **architecture-driven specification** as a prescription to help alleviate the difficulty of distributed system specification and design. Architecture-driven specification methods are advantageous because they exploit and re-use domain knowledge — know how built up from a previous history of solutions. This domain knowledge is often embodied in the forms of generic concepts, general ingredients, template-components, etc. In section 2.3 we reviewed **reference architectures** as examples of existing work that support architecture-driven specification and design. Chapter 4 realised our intention to construct a reference infra-structure to support the architecture-driven specification of distributed systems. Our infra-structure is unique because it supports specification using Extended LOTOS, and it places equal emphasis on both *performance* concerns and *functionality* concerns.

9.2.1.2 Performance specification

In distributed systems, the interplay between concurrency, asynchronous communication, spatial separation, etc. makes dealing with performance issues (such as time critical communications, adequate performance, probability of failures, etc.) very difficult (chapter 2). This difficulty elevates performance issues to an importance not found in non distributed systems. Consequently, specification languages for distributed systems need to have features for the expression of performance concerns. In the thesis, we decided to use LOTOS as our formal language for distributed systems specification (section 3.3). LOTOS is good at expressing functionality requirements but poor at expressing performance requirements. To remedy this, the thesis developed extensions to LOTOS for the specification of the performance concerns of quantitative timing (chapter 6), probability (chapter 7) and priority (chapter 8).

9.2.1.3 Practical application

For any theoretical work to be of use, it must have practical applications. The products of the theoretical work in this thesis are the infra-structure of architectural elements (chapter 4) and XL (chapters 6, 7 and 8). The case-study in chapter 5 demonstrates a practical application of this theoretical work.

The case-study in chapter 5 also fufills a more specific aspect of practical application: the application of (Extended) LOTOS to the CIM-OSA case-study has contributed to the existing body of knowledge and experience in the use of LOTOS in particular problem domains.

9.2.2 An infra-structure of architectural elements

To support the notion of architecture-driven specification, the thesis (chapter 4) builds an infra-structure of formalised architectural concepts and components.

This infra-structure is created with specification in mind. Its structure is hierarchical — founded upon very general *principles for description*, and rising to more specific *common architectural components*. XL representations are suggested for architectural elements; the higher level architectural elements are given graphical representations that help a reader see at-a-glance the structure of a specification. XL is used as the formal language for representing architectural elements because it supports the specification of performance concerns. Architectural components are defined for the specification of performance aspects, and these performance components are treated as equals with functionality and structuring components. The architectural components may combined to create either constraint-oriented or resource-oriented XL specifications.

9.2.3 Application of (Extended) LOTOS to CIM-OSA

Chapter 5 presented the CIM-OSA IIS (Computer Integrated Manufacturing — Open Systems Architecture Integrating Infrastructure) as a case-study of architecture-driven specification using XL.

CIM-OSA proved interesting for two reasons. Firstly, CIM-OSA contrasts with previous application domains for the use of LOTOS: we found that CIM-OSA is not a symmetric, layered communications architecture such as OSI; and CIM-OSA is more applied and specialized than the very general ODP architecture. Secondly, the CIM-OSA IIS is a distributed computing system which includes both functional and performance requirements. This made CIM-OSA a suitable candidate for testing the descriptive power of the performance features of XL.

In section 5.3 we showed how chapter 4's *common architecture components* could be used to build a skeleton architecture for the IIS. Then, in section 5.4, we focussed on one part of this skeleton to show how the *common architecture components* could be customized, and the performance features of XL used, to specify the SE_Service. We found that XL allowed quantitative timing, probabilistic and priority requirements to be expressed and composed easily. Furthermore, the classification of architectural components in section 4.4 provided guidelines for structuring the (de)composition of the IIS. The resulting XL specifications were organized in terms of architectural components, rather than solely in terms of specification language concepts. We found that the direct reflection of problem-domain structure in the specification made it easier to understand and navigate through the formal specification.

9.2.4 Extensions to the LOTOS language

The thesis develops **Extended LOTOS (XL)** — a formal, LOTOS based, language for the specification of distributed systems. XL is defined to be the integration of the three extensions to LOTOS we call **TLOTOS**, **PbLOTOS** and **PrLOTOS**. The work constituting these extensions is summarized in the following three subsections.

9.2.4.1 Extensions to LOTOS for quantitative timing

In chapter 6 we developed TLOTOS: LOTOS enhanced for the formal specification of quantitative timing concerns.

We exemplified the inadequacies of LOTOS with respect to the specification of quantitative timing concerns (section 6.2). Then we investigated, using a derivative of arc-timed Petri-Nets, the language facilities needed for the specification of timing requirements (section 6.3). A set of quantitative time features were distilled from the findings of this investigation, and a proposal was made to incorporate these into an extended version of LOTOS we called TLOTOS (section 6.5). TLOTOS was contrasted with other existing proposals in this area and found useful (section 6.4).

The syntax and semantics of TLOTOS were defined as extensions of the LOTOS syntax and semantics (section 6.5). TLOTOS semantics define a global, discrete clock which can be used to force events to occur at specific times (using *must* timing), and to measure the intervals between event occurrences. TLOTOS introduces *time-policies*: ASAP ('as soon as possible' corresponding to 'maximal progress semantics') and ALAP ('as late as possible) (section 6.3.9).

Section 6.6 looked at ways of mapping TLOTOS specifications to LOTOS. No satisfactory automatic means for doing this could be found. Nevertheless, this work proved useful as a basis for manually describing quantitative timing concerns in Standard LOTOS.

Appendix G extends the definitions of the LOTOS testing relations, and shows that extended versions of the testing relations yield sensible and intuitive results when applied to TLOTOS specifications.

9.2.4.2 Extensions to LOTOS for probability

In chapter 7 we developed PbLOTOS: LOTOS enhanced for the formal specification of probabilistic concerns.

We extended the definition of LTSs (Labelled Transition Systems) to define NP-LTSs (Non-deterministic and Probabilistic LTSs) and P-LTSs (Probabilistic LTSs) (section 7.3). NP-LTSs are LTSs which may contain both non-deterministic and probabilistic transitions. P-LTSs are LTSs which contain only probabilistic transitions.

NP-LTSs were used as a semantic model for PbLOTOS (section 7.3.3). PbLOTOS has a probabilistic choice operator for specifying probability distributions over a set of internal probability transitions.

We defined the *probabilization* pre-order (*prob*) as a formal implementation relation between NP-LTSs (section 7.4). "Probabilization" of an NP-LTS involves replacing non-deterministic transitions by probabilistic transitions. In this way, we can consider that an NP-LTS S describes a set of implementations $\{I | I\ prob\ S\}$ (P-LTSs).

We showed how to characterize the set of probabilistic implementations $\{I | I\ prob\ S\}$ as a set of simultaneous equations, where each equation describes the probability of an observable trace of S. The *free-terms* within the simultaneous equations generate the *set* of probabilistic implementations. These free-terms are used to range over the possible probabilizations of non-deterministic branches in S. We defined an algorithm $SimChar$ which, given an NP-LTS, generates a characterizing set of simultaneous equations (section 7.4.7).

In contrast to other probabilistic process algebras, we moved the emphasis away from the semantics of the PbLOTOS language, and instead placed many of the 'probabilistic concepts' in the associated theory of relations. The consequence of this is that the probabilistic aspects of the PbLOTOS semantics are simpler than the probabilistic aspects of the semantics of other process algebra, but the PbLOTOS theory for relations is consequently more complex.

We developed (section 7.5) a simple statistical testing framework for establishing whether a probabilistic implementation (a P-LTS) is a valid implementation of a probabilistic specification (an NP-LTS) according to the probabilization relation. We investigated if the "copying" and "minimum probability" assumptions could be used as a basis for interpreting test observations and making hypothesis decisions about probabilistic systems. However, we decided that these assumptions were unrealistic and abandoned them in favour of an established general purpose statistical inference method (χ^2). We considered PbLOTOS specifications to be mathematical objects, and their final realizations to be real world objects. Then we claimed total confidence in a hypothesis decision about the validity of a relation between two objects Q and P if these are PbLOTOS specifications (i.e. objects in the mathematical world). However, if at least one of these objects is a real world implementation, then we can have only some level of statistical confidence in our hypothesis decision.

9.2.4.3 Extensions to LOTOS for priority

In section 8.1 we developed PrLOTOS: LOTOS enhanced for the formal specification of priority concerns.

We defined a prioritized event to have a *priority-class* and *priority-value*. Where there is a choice between events from the same *priority-class*, the event with the highest *priority-value* will be fired. A choice between events from different *priority-classes* is rationalized to a non deterministic choice between the events with the highest *priority-value* in their respective *priority-class*. A choice between unprioritized events and prioritized events gives rise to non-deterministic choice. Prioritized event offers may synchronize with unprioritized event offers, thus prioritizing these unprioritized event offers through what we have called *association*.

9.3 Future work

We foresee two main themes of future work: the development of additional concepts and theories, and the development of software support tools.

Possibilities exist for defining new XL language constructs. For instance, new set operators (e.g. *setNotIncludingInterval*, *setNotEq*) could be invented for defining *inequality* over sets of *TimeSort*, in addition to the *equality* operators (e.g. *setInterval*, *setEQ*) described in section 6.5. New inequality operators would provide the power to express the quantitative times at which events are *not* permitted to occur. In a similar vein, new relational operators (e.g. $[< \mu]$, $[\leq \mu]$, $[\geq \mu]$, $[> \mu]$) could be defined for PbLOTOS, in addition to the existing $[= \mu]$ operator (section 7.3.3). Using these new relational operators, we could express probability requirements such as: 'message delivery will be *at least* 98% reliable', or 'the probability of failure will be *less than or equal to* 0.05'.

Completely new formal relations could be invented, or existing ones extended, to support XL development. Already, the thesis has invented (based on the notion of probabilization) a new implementation relation for PbLOTOS (section 7.4), and has extended the existing LOTOS testing relations for TLOTOS (appendix G).

A project known as TOPIC [TOP92] is currently developing verification methods and tools for Quality of Service (QoS) specification. TOPIC research includes investigating the formal treatment of time and probability in behaviour models (including LOTOS). This work is still at an early stage, but it will be interesting to see how this work compares with the work on time and probability described in this thesis.

Chapter 4 paves the way for the creation of a complete taxonomy of generic architectural components for distributed system specification and design. Chapter 4 has laid the foundation for this taxonomy by defining, and assigning XL representations to, the generic architectural components (such as the *timeout*, *FIFO* and *resource management components* in figure 4.12). Future work could assemble and provide XL templates for a collection of more specific components (such as the ODP *trader*, *configuration manager*, etc.).

The aim of recent ODP work [ISO93a, ISO93b] is to give LOTOS definitions to architectural concepts for distributed systems. It would be useful to analyse these emerging ODP LOTOS definitions and, if possible, align the architectural concepts and XL definitions developed for CIM-OSA with the emerging ODP LOTOS definitions. Aligning CIM-OSA architectural definitions with those of ODP (possibly basing the higher level CIM-OSA concepts upon the ODP basic modelling concepts) would make CIM-OSA systems instances of ODP systems, and allow CIM-OSA to take advantage of ODP work.

In addition to developing the concepts and theories described above, a second theme of future work is the development of software support tools. We visualize the development of tools that: support architecture-driven specification, through the rule-guided assembly of pre-designed domain-specific components; are visually-oriented, allowing their users to work in a graphical notation; and have a formal basis, generating XL code from the graphical design. Also, existing LOTOS syntax checkers, semantics analysers, simulators, etc. will have to be extended to support the quantitative timing, probability and priority features of the XL language. [WGW92] provides an example of what a simulation tool might look like for a specification language containing performance statements.

9.4 Concluding remarks

In summary, this thesis has shown that distributed systems can be effectively specified and analysed in an architecture-driven way using an extended form of the LOTOS formal specification language. An architecture framework was defined to support the architecture-driven specification of distributed systems. Functional issues and performance issues both play important rôles in distributed system specification. To reflect this, the architectural components have been defined using Extended LOTOS. Extended LOTOS was defined in this thesis as LOTOS enhanced with features for the formal specification of quantitative timing, probabilistic and priority concerns. CIM-OSA was used as a case-study for the application of architecture-driven specification and Extended LOTOS.

Bibliography

[Abr87] Samson Abramsky. Observation equivalence as a testing equivalence. *Theoretical Computer Science*, 53(2,3):225–241, 1987.

[AJ89] Heather Alexander and Valerie M. Jones. *Software Design and Prototyping using* me too. Prentice-Hall, New Jersey, 1989.

[Ald90] R. Alderden. COOPER – the compositional construction of a canonical tester. In *[Vuo89]*, pages 13–18, 1990.

[ANS86] ANSA. ANSA: Functional specification manual (release 1). Technical Report FS.37.1, Advanced Networked Systems Architecture, Architecture Projects Managment Limited, Poseidon House, Castle Park, CAMBRIDGE, U.K., May 1986.

[ANS89a] ANSA. ANSA: An engineer's introduction to the architecture. Technical Report TR.03.02, Advanced Networked Systems Architecture, Architecture Projects Managment Limited, Poseidon House, Castle Park, CAMBRIDGE, U.K., November 1989.

[ANS89b] ANSA. *The ANSA Reference Manual*. Architecture Projects Managment Limited, Poseidon House, Castle Park, CAMBRIDGE, U.K., 1989.

[AQ90] Arturo Azcorra and Juan Quemada. Proposal for the introduction of time in LOTOS. Technical Report Lo/WP3/T3.3/UPM/N0013/V01, LOTOSPHERE (ESPRIT 2304), January 1990.

[BAPM83] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Inf.*, 20:207–226, 1983.

[Bar87] Howard Barringer. Up and down the temporal way. *The Computer Journal*, 30(2):134–148, 1987.

[BB86] F. P. M. Biemans and P. Blonk. On the formal specification and verification of CIM architecture using LOTOS. *Computors in Industry*, 7:491–504, 1986.

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1), 1987.

[BC89] Tommaso Bolognesi and M. Caneve. SQUIGGLES: A tool for the analysis of LOTOS specifications. In *[Tur88a]*, pages 201–216, 1989.

[BdMS89] P. Bohm, J. de Meer, and P. Schoo. Perlon persistency checker for data type definitions. In *[BSV89]*, pages 285–302, 1989.

[Bee89] Dirk Beekman. CIM-OSA: Computer integrated manufacturing — open systems architecture. *Int. Journ. Computer Integrated Manufacturing*, 2(2):94–105, 1989.

[Bie89] F. P. M. Biemans. *A Reference Model for Manufacturing Planning and Control*. PhD thesis, University of Twente, NL, 1989.

[BIM88] Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced: Preliminary report. In *Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California*, pages 229–239, 1988.

[BJ78] D. Bjoner and C. B. Jones. *The Vienna Development Method: The Meta-Langaugr*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[BK84] Howard Barringer and Rudolph Kuiper. Hierarchical development of concurrent systems in a temporal logic framework. *Seminar on Concurrency. Lecture Notes in Computer Science*, 129:35–61, 1984.

[BKP84] Howard Barringer, Rudolph Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *16th ACM TOC*, pages 51–63, 1984.

[BKP85] Howard Barringer, Rudolph Kuiper, and A. Pnueli. A compositional approach to a CSP like language. In *Proc. IFIP Working Conf.: The Role of Abstract Models in Information Processing, Vienna*, 1985.

[BL91] Tommaso Bolognesi and F. Lucidi. LOTOS like process algebras with urgent or timed interactions. In *[PR91]*, pages 255–270, 1991.

[Bla89] Stewart Black. Objects and LOTOS. Technical report, Hewlett-Packard Laboratories, Stoke Gifford, Bristol, BS12 6QZ, England, Great Britain, October 1989.

[BM89] Bard Bloom and Albert R. Meyer. *A Remark on Bisimulation between Probabilistic Processes*, volume 363 of *Lecture Notes in Computer Science (Logic at Botik '89), Broy, M. and Jones, C. B. (eds)*, pages 26–40, 1989.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BNO88] David W. Bustard, M. T. Norris, and R. A. Orr. A pictorial approach to the animation of process-oriented formal specifications. *Software Engineering Journal*, 3(4):114–118, July 1988.

[Bog90] Kees Bogaards. *A Methodology for the Architectural Design of Open Distributed Systems*. PhD thesis, University of Twente, NL, 1990.

[Bol90] Tommaso Bolognesi. Timed LOTOS: Which way to go? In *British Telecom – British Computer Society/FACS Group Meeting on LOTOS – London*, September 1990.

[Boo87] Grady Booch. *Software Engineering with Ada*. Cummings-Benjamin, second edition, 1987.

[Bri88a] Ed Brinksma. *On the Design of Extended LOTOS*. PhD thesis, Uni. of Twente, NL, 1988.

[Bri88b] Ed Brinksma. A theory for the derivation of tests. In *Proc. Eight Int. Symp. on Protocol Specification, Testing and Verification, Atlantic City, New Jersey, USA*, 1988.

[Bri91] Ed Brinksma. What is the method in formal methods? In *[PR91]*, pages 33–50, 1991.

[BS81] H. J. Burkhardt and S. Schindler. Structuring principles of the communication architecture of open systems – a systematic approach. 1981.

[BS86] Ed Brinksma and Giuseppe Scollo. Formal notions of implementation and conformance in LOTOS. Technical report, Twente University of Technology, NL, December 1986.

[BSV89] Ed Brinksma, Giuseppe Scollo, and Chris A. Vissers, editors. *Proc. of the 9th Int. Symp. on Protocol Specification, Testing and Verification*, Amsterdam, 1989. North-Holland.

[BV89] Stewart Black and Patrick Viollet. CIM OSA: Activity control service description in LOTOS. Technical report, Hewlett-Packard Laboratories, Filton Road, Stoke Gifford, Bristol BS12 6QZ, U.K., October 1989.

[BWN+88] David W. Bustard, Adam C. Winstanley, M. T. Norris, R. A. Orr, and S. Patel. Graphical views of process-oriented specifications. In *[Tur88a]*, pages 143–156, September 1988.

[CC83] G. M. Clarke and D. Cooke. *A Basic Course in Statistics*. Edward Arnold, second edition, 1983.

[CCI88a] CCITT. Abstract service definition conventions. International Consultative Committee on Telegraphy and Telephony, 1988. Recommendation X.407.

[CCI88b] CCITT. Directory system, 1988. X.500 series Recommendations.

[CCI88c] CCITT. Message handling systems, 1988. X.400 series Recommendations.

[CCI90a] CCITT. DAF: Design method. Technical Report JTC1/SC21 WG7/N223, International Consultative Committee on Telegraphy and Telephony, 1990.

[CCI90b] CCITT. DAF: Requirements for specification techniques and languages. Technical Report JTC1/SC21 WG7/N226, International Consultative Committee on Telegraphy and Telephony, 1990.

[CCI90c] CCITT Q19/VII. DAF: Interpreting modelling concepts in LOTOS. Technical Report WG7/N225, International Consultative Committee on Telegraphy and Telephoney, 1990.

[CCI92] CCITT. Specification and description language. International Consultative Committee on Telegraphy and Telephony, 1992. Recommendation Z.100.

[CD88] George F. Coulouris and Jean Dollimore. *Distributed Systems: Concepts and Design*. International Computer Science Series. Addison Wesley, 1988.

[CdC91] Jean-Pierre Courtiat and Joao Coelho da Costa. A LOTOS based calculus with true concurrency semantics. In *[PR91]*, pages 559–574, 1991.

[CIM89a] CIM-OSA. FRB series on "business services complex (B)". Technical Report CIM-OSA C5-4xxx Series, WP-H, CIM-OSA, Esprit 688, 1989.

[CIM89b] CIM-OSA. FRB series on "system wide exchange (SE)". Technical Report CIM-OSA C5-12xx Series, WP-J, CIM-OSA, Esprit 688, 1989.

[CIM89c] CIM-OSA. *Open System Architecture for CIM*. Number ISBN 3-540-52058-9. Springer-Verlag, 1989.

[CIM90a] CIM-OSA. Architecture description — CIM-OSA AD 1.1. Technical Report R0391/1, CIM-OSA, Esprit 688, 1990.

[CIM90b] CIM-OSA. FRB series on "machine front end (MF)". Technical Report CIM-OSA C5-33xx Series, WP-G, CIM-OSA, Esprit 688, 1990.

[CIM90c] CIM-OSA. FRB series on "system wide data (SD)". Technical Report CIM-OSA C5-21xx Series, WP-F, CIM-OSA, Esprit 688, 1990.

[CIM90d] CIM-OSA. Management overview — CIM-OSA AD 1.1. Technical Report R0391/0, CIM-OSA, Esprit 688, 1990.

[CIM90e] CIM-OSA. System wide capability of the IIS services. Technical Report CIM-OSA OI-0002, WP-H, CIM-OSA, Esprit 688, 1990.

[CJ92] Robert G. Clark and Valerie M. Jones. Use of LOTOS in the formal development of an OSI protocol. *Computer Communications*, 15(2):86–92, March 1992.

[Cla90] Robert G. Clark. Using LOTOS in the object-based development of embedded systems. In *[RC90]*, pages 307–319, 1990.

[CM91] John M. Carroll and Thomas P. Moran. Introduction to this special issue on design rationale. *Human-Computer Interaction*, 6(3&4):197–200, 1991.

[CO89] CIM-OSA. FDT task force minutes, December 1989. Esprit.

[Coc52] W. G. Cochran. The χ^2 test of goodness of fit. *Ann. Math. Stat.*, 23:315–345, 1952.

[CPW86] B. Cohen, D. H. Pitt, and J. C. P. Woodcock. The importance of time in the specification of OSI protocols: An overview and brief survey of the formalisms. Technical Report ISSN 0262-5369, National Physical Laboratory, Teddington, Middlesex, TW11 0LW, U.K., November 1986.

[CRS89] Elspeth Cusack, Steve Rudkin, and Chris Smith. An object-oriented interpretation of LOTOS. Technical report, British Telecom Research & Technology, St. Vincent House, 1, Cutler Street, Ipswich, IP1 1UX, United Kingdom., October 1989.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Cyp78] R. J. Cypser. *Communications Architecture for Distributed Systems*. The Systems Programming Series. Addison-Wesley, first edition, 1978.

[CYYW89] 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 *[Vuo89]*, pages 31–36, 1989.

[DH70] N. M. Downie and R. W. Heath. *Basic Statistical Methods*. Harper & Row Publishers, third edition, 1970.

[Dia91] M. Diaz. Formal description techniques based on state approaches. FORTE'91, Forth International Conference on: Formal Description Techniques, 1991. (tutorial).

[EH86] E. A. Emerson and J. Y. Halperm. "sometimes" and "not never" revisited. *Journ. ACM*, 33(1):151–178, January 1986.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, volume 6 of *EATCS Monographs on Theoretical Computer Science, Brauer, W. and Rozenberg, G. and Saloman, A. (eds)*. Springer-Verlag, 1985.

[ERP90] José Manuel Martin Espinosa, José Miguel Robles Roman, and Luis Fuertes Prieto. Concurrent modelling in LOTOS as a solution to real time problems. In *[QMV90]*, 1990.

[ESP87] ESPRIT. Computer integrated manufacturing (CIM). In *Esprit: The First Phase: Progress and Results 1986*. ISBN 92-825-6916-0. Luxemburg: Office for Publications of the European Communities, 1987.

[ESP88] ESPRIT. Computer integrated manufacturing (CIM). In *European Strategic Programme for Research and Development in Information Technology. 1987 Annual Report*. ISBN 92-825-83791-1. Luxemburg: Office for Publications of the European Communities, 1988.

[Eur88] European Computer Manufacturers Association. Information processing systems — basic remote procedure call (RPC) using OSI remote operations. International Organization for Standardization, 1988. ISO/IEC DIS 10148.

[FA88] David Freestone and S. S. Aujla. Specifying ROSE in LOTOS. In *[Tur88a]*, pages 231–245, 1988.

[Fel68] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. John Wiley & Sons, Inc., third edition, 1968.

[FGL90] Alessandro Fantechi, Stefania Gnesi, and C. Laneve. An expressive temporal logic for basic LOTOS. In *[QMV90]*, pages 261–276, 1990.

[FGM90] Alessandro Fantechi, Stefania Gnesi, and Gianluca Mazzarini. How expressive are lotos behaviour expressions? In *[QMV90]*, pages 17–32, 1990.

[Fid90] Colin J. Fidge. A LOTOS interpreter for simulating real-time behaviour. In *[QMV90]*, 1990.

[Fid92] Colin J. Fidge. Process algebra traces augmented with causal relationships. In *[PR91]*, pages 527–541, 1992.

[Gib93] J. Paul Gibson. *Formal Object Oriented Development of Software Systems using LOTOS*. PhD thesis, Uni. of Stirling, Scotland, 1993.

[GJS90] Alessandro Giacalone, Chi-Chang Jou, and Scott A. Smolka. Algebraic reasoning for probabilistic concurrent systems. In M. Broy and C. B. Jones, editors, *Proc. of the IFIP Working Group 2.2/2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 443–458. Elsevier Science Publishers B.V. (North Holland), 1990.

[Got92] Reinhard Gotzhein. *Open Distributed Systems: On Concepts, Methods, and Design from a Logical Point of View*. PhD thesis, Uni. of Hamburg, 1992.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. The temporal analysis of fairness. In *7th ACM POPL*, pages 163–173, 1980.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Gui90] Raymonde Guindon. Knowledge exploited by experts during software system design. *International Journal of Man-Machine Studies*, 33(3):279–304, 1990.

[Hal88] Fred Halsall. *Data Communications, Computer Networks and OSI*. Addison Wesley, second edition, 1988.

[Ham89] Alan G. Hamilton. *The Professional Programmers Guide to Prolog*. Pitman, 1989.

[Han90] Hans Hansson. Modelling timeouts and unreliable media with a timed probabilistic calculus. In *[PR91]*, pages 67–82, 1990.

[HdR89] J. J. M. Hooman and William P. de Roever. Design and verification in real-time distributed computing: an introduction to compositional methods. In *Proceedings of the 9th IFIP WG 6.1 Int. Symp. on Protocol Specification, Testing, and Verification, (Editors: Ed Brinksma, Giuseppe Scollo, Chris A. Vissers), North-Holland*, 1989.

[HJ89] Hans Hansson and Bengt Jonsson. A framework for reasoning about time and reliability. In *Proc. 10th IEEE Real-Time Systems Symp., Santa Monica*, pages 102–111. Computer Soc. Press, 1989.

[HJ90] Hans Hansson and Bengt Jonsson. A calculus for communicating systems with time and probabilities. In *Proc. 11th IEEE Real-Time Systems Symp., Orlando Florida*, pages 278–287. Computer Soc. Press, 1990.

[HM80] Matthew Hennessy and Robin Milner. *On Observing Nondeterminism and Concurrency*, volume 85 of *Lecture Notes in Computer Science (Automata, Languages and Programming), de Bakker, J. W. and van Leeuwen, J. (eds)*, pages 299–309. 1980.

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computing Machinery*, 32(1):137–161, January 1985.

[HO83] B. T. Hailpern and S. S. Owicki. Modular verification of computer communication protocols. *IEEE Trans. Comms.*, 31(1):56–68, 1983.

[Hoa83] C. A. R. Hoare. Notes on communicating sequential processes. Technical Report Technical Monograph PRG-33, Oxford Uni. Computing Laboratory, August 1983.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[Hog90] Dieter Hogrefe. Conformance testing based on formal methods. In *[QMV90]*, pages 207–222, 1990.

[HU79] John E. Hopcroft and Jeffery D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.

[ISO82] ISO. Specification for computer programming language pascal. International Organization for Standardization, 1982. 7185.

[ISO84] ISO. Information processing systems – open systems interconnection – basic reference model. International Organization for Standardization, 1984. 7498.

[ISO88a] ISO. Information processing systems – text and office systems – distributed-office-application model. part 1: General model. International Organization for Standardization, March 1988. ISO/IEC JTC1/SC18/WG4 N865.

[ISO88b] ISO. Information processing systems – text and office systems – distributed-office-application model. part 2: Referenced data transfer. International Organization for Standardization, March 1988. ISO/IEC JTC1/SC18/WG4 N866.

[ISO89a] ISO. Information processing systems – open systems interconnection – ESTELLE – a formal description technique based on an extended state transition model. International Organization for Standardization, 1989. 9074.

[ISO89b] ISO. Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour. International Organization for Standardization, 1989. 8807.

[ISO89c] ISO. Working document on topic 4.1 – structures and functions. International Organization for Standardization, December 1989. ISO/IEC JTC1/SC21 N4022.

[ISO89d] ISO. Working document on topic 6.1 – modelling techniques and their use in ODP. International Organization for Standardization, May 1989. ISO/IEC JTC1/SC21/WG7.

[ISO89e] ISO. Working document on topic 8.1 – draft basic reference model of open distributed processing – part ii. International Organization for Standardization, December 1989. ISO/IEC JTC1/SC21 N4025.

[ISO90a] ISO. Information processing systems – open systems interconnection – formal description in LOTOS of the connection-oriented transport service. International Organization for Standardization, 1990. TR 10023.

[ISO90b] ISO. Information processing systems – open systems interconnection – formal description in LOTOS of the connection-oriented transport protocol. International Organization for Standardization, 1990. TR 10024.

[ISO90c] ISO. Information processing systems – open systems interconnection – formal description in LOTOS of the connection-oriented session service. International Organization for Standardization, 1990. TR 9571.

[ISO90d] ISO. Information processing systems – open systems interconnection – formal description in LOTOS of the connection-oriented session protocol. International Organization for Standardization, 1990.

[ISO91] ISO. Information processing systems – open systems interconnection – guidelines for the application of ESTELLE, LOTOS, and SDL. Technical report, International Organization for Standardization, 1991. TR 10167, Kenneth J. Turner (ed).

[ISO92a] ISO. LOTOS specification of the trader. International Organization for Standardization, 1992. ISO/IEC JTC1/SC21/WG7 N743 Annex C.

[ISO92b] ISO. Use of formal specification techniques for ODP. International Organization for Standardization, 1992. ISO/IEC JTC1/SC21/WG7 N753.

[ISO93a] ISO. Draft – ODP architectural semantics using LOTOS. International Organization for Standardization, 1993. ISO/IEC JTC1/SC21/WG7, Richard Sinott (ed).

[ISO93b] ISO. Draft – ODP architectural semantics using Z. International Organization for Standardization, 1993. ISO/IEC JTC1/SC21/WG7, Richard Sinott (ed).

[ISO93c] ISO. Draft answer to question Q1/48.6 – E-LOTOS – proposed extensions to LOTOS. International Organization for Standardization, SC21 WG1, June 1993.

[Jac83] M. A. Jackson. *System Design*. Prentice-Hall, 1983.

[JBD89] Albert Jones, Edward Barkmeyer, and Wayne Davis. Issues in the design and implementation of a system architecture for computer integrated manufacturing. *Int. J. Computer Integrated Manufacturing*, 2(2):65–76, 1989.

[JC90] Valerie M. Jones and Robert G. Clark. LOTOS specification of the OSI CCR protocol. Esprit Project 2304, Commission of the European Communities, Brussels, 1990. LOTOSPHERE, Lo/WP3/T3.1/UST/N0003/V04.

[Joh89] S. G. Johnson. SPIDER – service and protocol interactive development environment. In *[Tur88a]*, pages 67–71, 1989.

[Kay93] James W. Kay. Statistics lecture notes. Dept. of Mathematics, Uni. of Stirling, Scotland, 1993.

[KP85] Ed Knepley and Robert Platt. *Modula-2 Programming*. Reston Publishing Company, Inc., Reston, Virginia., 1985.

[Lam77] Leslie Lamport. Proving the correctness of multi process programs. *IEEE Transactions on Soft. Eng.*, SE-3(2):21–37, March 1977.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam80] Leslie Lamport. 'sometime' is sometimes 'not never'. In *A tutorial on the Temporal Logic of Programs in Proc. of the 7th Ann. Symp. on Principles of Programming Languages (ACM SIGACT-SIGPLAN)*, January 1980.

[Lam83] Leslie Lamport. What good is temporal logic. In *In Proc. of IFIP Conf. on Information Processing 1983*, pages 657–668. North-Holland, 1983.

[Led90] Guy Leduc. *On the Role of Implementation Relations in the Design of Distributed Systems using LOTOS*. PhD thesis, Uni. of Liège, 1990.

[Led91a] Guy Leduc. Conformance relation, associated equivalence, and minimum canonical tester in LOTOS. Technical report, Université de Liège, August 1991.

[Led91b] Guy Leduc. An upward compatible timed extension to LOTOS. In *[PR91]*, pages 223–238, 1991.

[Lin89] Peter F. Linington. Why OSI? *Computer Networks and ISDN Systems*, 17(4 & 5):287–290, October 1989.

[Lin91] Peter F. Linington. Introduction to the open distributed processing basic reference model. In *1st International Workshop on Open Distributed Processing, Berlin*, October 1991.

[LL91] Jintae Lee and Kum-Yew Lai. What's in design rationale? *Human-Computer Interaction*, 6(3&4):251–280, 1991.

[LOBF88] L. Logrippo, A. Obaid, J. P. Briand, and M. C. Fehri. An interpreter for LOTOS, a specification language for distributed systems. *Software - Practice and Experience*, 18(4):365–385, April 1988.

[LPS81] B. W. Lampson, M. Paul, and H. J. Siegert, editors. *Distributed Systems Architecture and Implementation*. Springer-Verlag, second edition, 1981.

[LS89] Kim G. Larsen and Arne Skou. Bisimulation through probabilistic testing. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas*, pages 344–352, 1989.

[Mar89] A. K. Marshall. Introduction to LOTOS tools. In *[vEVD89]*. North-Holland, 1989.

[Mas80] Mascot Suppliers Association, Computer Standards Section, Room L303, Royal Signals and Radar Establishment, St. Andrews Rd., Malvern, Worcester, England. *The Official Handbook of MASCOT*, December 1980.

[May89] Thomas Mayr. Specification of object oriented systems in LOTOS. *Formal Description Techniques*, pages 107–119, 1989.

[MBB90] Ashley McClenaghan, Daniel Boisson, and Stewart Black. LOTOS specification of the CIM-OSA IIS SD-Service-Definition. Technical report, CIM-OSA, Esprit 688, 1990.

[MC93] Ana M. D. Moreira and Robert G. Clark. Using rigorous object-oriented analysis. Technical Report TR111, Department of Computing Science and Mathematics, University of Stirling, Stirling, Scotland., August 1993.

[McC90a] Ashley McClenaghan. Aspects of the formal specification of the CIM-OSA IIS architecture. Technical report, CIM-OSA, Esprit 688, 1990.

[McC90b] Ashley McClenaghan. On the specification of the CIM-OSA system wide exchange. Technical report, CIM-OSA, Esprit 688, 1990.

[McC91a] Ashley McClenaghan. Experience of using LOTOS within the CIM-OSA project. In *[PR91]*, pages 113–120, 1991.

[McC91b] Ashley McClenaghan. Mapping time extended LOTOS to standard LOTOS. In *[PR91]*, pages 239–254, 1991.

[MdM89] J. A. Manas and T. de Miguel. From LOTOS to C. In *[Tur88a]*, pages 79–84, 1989.

[Mey87] Bertrand Meyer. Eiffel: Programming for reusability and extendability. *ACM SIGPLAN Notices*, 22:85–94, 1987.

[Mey88a] Bertrand Meyer. Eiffel: A language and environment for software engineering. *The Journal of Systems and Software*, 8:199–246, 1988.

[Mey88b] Bertrand Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

[Mey93] Bertrand Meyer. Systematic concurrent object-oriented programming. In *13th IFIP Symp. on Protocol Specification, Testing and Verification, Liège, André Danthine, Guy Leduc and Pierre Wolper (eds)*, 1993.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. 1980.

[Mil81] Robin Milner. *A modal characterisation of observable machine-behaviour*, volume 112 of *Lecture Notes in Computer Science (CAAP'81), Astesiano, E. and Bohm, C. (eds)*, pages 25–34. 1981.

[Mil83] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[MP81] Z. Manna and A. Pnueli. The verification of concurrent programs: The temporal framework. In *The Correctness Problem in Computer Science, R. S. Boyer, J. S. Moore (eds)*, pages 215–273, 1981.

[MTCM80] A. Malhotra, J. C. Thomas, J. M. Carroll, and L. A. Miller. Cognitive processes in design. *International Journal of Man-Machine Studies*, 12(2):119–140, 1980.

[MV89] E. Madelaine and D. Vergamini. AUTO: A verification tool for distributed systems using reduction of finite automata networks. In *[Vuo89]*, pages 79–84, 1989.

[Nel81] Bruce Jay Nelson. *Remote Procedure Call*. PhD thesis, Carnegie-Mellon University, May 1981.

[NH82] R. M. Needham and A. J. Herbert. *The Cambridge Distributed Computing System*. Addison-Wesley, 1982.

[OF88] Koji Okada and Kokichi Futatsugi. Supporting the formal description process for communication protocols by an algebraic specification language OBJ2. In *Proc. of the 2nd Int. Symp. on Interoperable Information Systems, INTAP*, pages 127–134, 1988.

[OIF91] Koji Okada, Masaki Ishigamori, and Kokichi Futatsugi. Systematic construction of services from protocol specifications by a parameterized description according to the OSI layered structure. In *Proc. of the 6th Joint Workshop on Communication (JWCC-6) Fukuoka, Japan*, July 1991.

[PAN89] PANGLOSS. Specification of the OSI connection-less internet protocol. Technical report, Esprit Project 890, Commission of the European Communities, Brussels, 1989.

[Pec92] Charles Pecheur. Using LOTOS for specifying the CHORUS distributed operating system kernel. *Computer Communications*, 15(2):93–102, March 1992.

[Pet62] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut fur Instrumentelle Mathematik, Bonn, FRG, 1962.

[Pet81] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[Pir91] Luís Ferreira Pires. The lotosphere design methodology: Basic concepts. Technical report, LOTOSPHERE, 1991. Lo/WP1/T1.1/N0045/V02.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Denmark, 1981.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. of the 19th IEEE Annual Symp. on Foundations of Computer Sci.*, pages 46–57, 1977.

[Pnu81] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Sci.*, 13:45–60, 1981.

[PR91] Ken Parker and Gordon Rose, editors. *FORTE'91, Fourth International Conference on: Formal Description Techniques*, Sydney, Australia, 1991. Elsevier Science Publishers B.V.

[QAF90] J. Quemada, A. Azcorra, and D. Frutos. TIC: A timed calculus for LOTOS. In *[Vuo89]*, pages 195–209, 1990.

[QMV90] Juan Quemada, Jose Manas, and Enrique Vazquez, editors. *FORTE'90, The IFIP Third International Conference on: Formal Description Techniques*, Madrid, Spain, November 1990. Elsevier Science Publishers B.V. (North-Holland).

[QPF89] Juan Quemada, Santiago Pavon, and Angel Fernandez. Transforming LOTOS specifications with LOLA: The parameterized expansion. In *[Tur88a]*, pages 45–54, 1989.

[RC90] Charles Rattray and Robert G. Clark, editors. *The Unified Computation Laboratory – Modelling, Specifications, and Tools*, Held at the University of Stirling, Scotland, July 1990. Institute of Mathematics and its Applications, Clarendon Press, Oxford.

[RM87] M. Rozier and L. Martins. The chorus distributed operating system: some design issues. *Distributed Operating Systems. Theory and Practice*, F28:261–287, 1987.

[ROS89a] ROSA. RACE open services architecture: Architectural workpackage, WP3 – outline of ROSA architecture. Technical Report 3nd Deliverable, 08/BTR/425/DS/B/003/b1, Malcolm Key (ed), RACE Open Services Architecture, R1088, November 1989.

[ROS89b] ROSA. RACE open services architecture: Object definition workpackage – object-oriented techniques for ROSA. Technical Report 1st Deliverable, 08/BTR/425/DS/B/001/b1, Malcolm Key (ed), RACE Open Services Architecture, R1088, July 1989.

[ROS89c] ROSA. RACE open services architecture: Service specification workpackage – specifying services using objects. Technical Report 2nd Deliverable, 68/BTR/425/DS/B/002/b1, Malcolm Key (ed), RACE Open Services Architecture, R1088, September 1989.

[ROS89d] ROSA. RACE open services architecture: Tool support workpackage, WP4 – outline requirements for tool support. Technical Report 4nd Deliverable, 68/BTR/425/DS/B/004/b1, Malcolm Key (ed), RACE Open Services Architecture, R1088, November 1989.

[Rud92] Steve Rudkin. Inheritance in LOTOS. In *[PR91]*, pages 409–424, 1992.

[Sad90] F. Sadoun. LOTOS specification of the OSI CCR service. Technical Report Lo/WP3/T3.1/SYS/N0007/V02, Esprit Project 2304, Commission of the European Communities, Brussels, 1990.

[SB87] Jurgen Suppan-Borowka. Planning the use of MAP. In *International Open Systems 87: Proceedings of the International Conference held in London*, volume 1, March 1987.

[SBN84] Michael D. Schroeder, Andrew D. Birrell, and Roger M. Needham. Experience with grapevine: The growth of a distributed system. *ACM Transactions on Computer Systems*, 2(1):3–23, February 1984.

[SK87] Morris Sloman and Jeff Kramer. *Distributed Systems and Computer Networks*. Prentice Hall International Series in Computer Science. Prentice-Hall, 1987.

[SMC74] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13:115–139, 1974.

[Spi89] J. M. Spivey. *The Z Notation*. Prentice-Hall, New Jersey, 1989.

[Ste91] Jean-Bernard Stefani. Open distributed processing: The next target for the application of formal description techniques. In *[QMV90]*, pages 427–442, 1991.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[Tan81] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall Software Series. Prentice-Hall, 1981.

[Toc89] Alastair J. Tocher. Formal support for the development of distributed systems. Technical Report ANSA RC.97.0, Advanced Networked Systems Architecture, Architecture Projects Managment Limited, Poseidon House, Castle Park, CAMBRIDGE, U.K., November 1989.

[Toc90] Alastair J. Tocher. Towards a theory of objects. Technical Report ANSA RC.066.02, Advanced Networked Systems Architecture, Architecture Projects Managment Limited, Poseidon House, Castle Park, CAMBRIDGE, U.K., May 1990.

[TOP92] TOPIC. TOPIC: Toolset for protocol and advanced service verification in ibc environments, 1992. RACE Proposal 13231.

[TS93] Kenneth J. Turner and Richard O. Sinnott. DILL: Specifying digital logic in LOTOS. In *FORTE'93, The IFIP 6th International Conference on: Formal Description Techniques*, 1993.

[Tur85] D. Turner. *Miranda: A Non-Strict Functional language with Polymorphic Types*, volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[Tur87] Kenneth J. Turner. An architectural semantics for LOTOS. In *Proc. of the IFIP WG 6.1 Seventh Int. Conf. on Protocol Specification, Testing, and Verification*, pages 15–28, May 1987.

[Tur88a] Kenneth J. Turner, editor. *FORTE'88, The IFIP First International Conference on: Formal Description Techniques*, Stirling, Scotland, September 1988. Elsevier Science Publishers B.V. (North-Holland).

[Tur88b] Kenneth J. Turner. PANGLOSS reference architecture development strategy. Esprit 890, August 1988. PANGLOSS/AT/UST/N005.

[Tur89a] Kenneth J. Turner. *The Formal Specification Language LOTOS: A Course For Users*. Department of Computing Science, University of Stirling, Stirling, August 1989.

[Tur89b] Kenneth J. Turner. A LOTOS-based development strategy. In *[Vuo89]*, pages 157–174, November 1989.

[Tur89c] Kenneth J. Turner. A LOTOS case study: Specification of the OSI connection-oriented network service. In *OTC Workshop on Formal Techniques, Sydney*, July 1989.

[Tur90] Kenneth J. Turner. Template-based specification in LOTOS. Technical report, Department of Computing Science and Mathematics, University of Stirling, Stirling, Scotland., 1990.

[Tur91] Kenneth J. Turner. The role of architecture in formalism. In *[PR91] (tutorial)*, 1991.

[Tur93a] Kenneth J. Turner. An engineering approach to formal methods. In *13th IFIP Symp. on Protocol Specification, Testing and Verification, Liège, André Danthine, Guy Leduc and Pierre Wolper (eds)*, 1993.

[Tur93b] Kenneth J. Turner. Formal specification and design with LOTOS. (In preparation), 1993.

[Tur93c] Kenneth J. Turner. *Using Formal Description Techniques: An Introduction to ESTELLE, LOTOS and SDL*. John Wiley & Sons, Inc., first edition, 1993.

[TvR85] A. S. Tanenbaum and R. van Renesse. Distributed operating systems. *Computing Surveys*, 17(4):419–470, 1985.

[TvS92] Kenneth J. Turner and Marten van Sinderen. OSI specification styles for LOTOS. In *Proc. of the 3rd LOTOSPHERE Workshop, Pisa*, pages 5/1–22, September 1992.

[vE88] Peter H. J. van Eijk. *Software tools for the Specification Language LOTOS*. PhD thesis, Twente University of technology, Enschede, Netherlands, 1988.

[vE89] Peter H. J. van Eijk. LOTOS tools based on the cornell synthesizer generator. In *[BSV89]*, 1989.

[vEE91] Peter H. J. van Eijk and H. Eertink. Design of the LotosPhere symbolic LOTOS simulator. In *[QMV90]*, 1991.

[vEVD89] Peter H. J. van Eijk, Chris A. Vissers, and Michel Diaz, editors. *The Formal Description Technique LOTOS*. North-Holland, 1989.

[vG89] Joost J. van Griethuysen. Open distributed processing (ODP). *Ninth IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification*, June 1989.

[vGSST90] Rob van Glabbeek, Scott A. Smolka, Bernhard Steffen, and Chris M. N. Tofts. Reactive, generative, and stratified models of probabilistic processes. In *Proc. of the 5th IEEE Int. Symp. on Logic in Computer Science*, pages 130–141, 1990.

[vH89] Wilfried H. P. van Hulzen. Object-oriented specification style in LOTOS. Technical Report Lo/WP1/T1.1/RNL/N0002, European LOTOSPHERE Consortium, Esprit 2304, July 1989.

[VH90] Willemien Visser and Jean Michel Hoc. *Expert Software Design Strategies*, pages 235–249. Psychology of Programming (T. Green *et al.* (eds)). Academic Press, 1990.

[vHTZ90] Wilfried H. P. van Hulzen, Paul A. J. Tilanus, and Han Zuidweg. LOTOS extended with clocks. In *[Vuo89]*, pages 179–193, 1990.

[Vio90] Patrick Viollet. LOTOS guidelines for the IIS. Technical Report R0373/0, CIM-OSA, Esprit 688, 1990.

[Vis90] Willemien Visser. More or less following a plan during design: opportunistic deviations in specification. *International Journal of Man-Machine Studies*, 33(3):247–278, 1990.

[vS90] Marten van Sinderen. Generic service and protocol structures. In *ESPRIT Conference '90, Brussels*, Uni. of Twente, NL, 1990.

[VSvS88] Chris A. Vissers, Giuseppe Scollo, and Marten van Sinderen. Architecture and specification style in formal descriptions of distributed systems. In *Proc. of the IFIP WG 6.1 Eight Int. Conf. on Protocol Specification, Testing, and Verification*. Elsevier Science Publishers B.V. (North-Holland)., 1988.

[VSvSB90] Chris A. Vissers, Giuseppe Scollo, Marten van Sinderen, and Ed Brinksma. On the use of specification styles in the design of distributed systems. University of Twente, NL, 1990.

[Vuo89] Son T. Vuong, editor. *The IFIP TC/WG 6.1 Second International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols, FORTE '89*, Vancouver, Canada, 1989. North-Holland.

[Wal93] Bernd Walter. Timed petri-nets for modelling and analyzing protocols with real-time characteristics. In *Proceedings of the IFIP 3rd International Workshop on Protocol Specification, Testing, and Verification, Harry Rudkin, Colin H. West (eds), North-Holland*, pages 149–159, 1993.

[WB87] Steve Wibur and Ben Bacarisse. Building distributed systems with remote procedure calls. Technical report, Dept. of Computer Science, University College London, 1987.

[WB89] D. Wolz and P. Bohm. Compilation of LOTOS data type specifications. In *[BSV89]*, pages 187–202, 1989.

[WB91] Adam C. Winstanley and David W. Bustard. EXPOSE: an animation tool for process-oriented specifications. *Software Engineering Journal*, 6(6):114–118, November 1991.

[WBL90] Clazien D. Wezeman, S. Batley, and James A. Lynch. Formal methods to assist conformance testing — a case study. In *[QMV90]*, pages 157–174, 1990.

[Wez89] Clazien D. Wezeman. The CO-OP method for the compositional derivation of conformance testers. In *Proc. Ninth IFIP WG 6.1 Int. Symp. on Protocol Specification, Testing, and Verification*, 1989.

[WGW92] Jorg Wolf Gunther and Adam Wolisz. SIMTIS — a simulation package for the performance evaluation of communication protocols specified as timed interacting systems. 1992.

[Win92] Adam C. Winstanley. *The Elucidation of Process-Oriented Specifications*. PhD thesis, Queen's Uni., Belfast, 1992.

[WvHR90] Ing Widya, Gert Jan van Heijden, and Francis Riddoch. LOTOS specification of the TP protocol. Technical Report Lo/WP3/T3.1/xxx/N0020/V02, Esprit Project 2304, Commission of the European Communities, Brussels, 1990.

[YC79] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, 1979.

[You89] Edward Yourdon. *Modern Structured Analysis*. Prentice-Hall, 1989.

Appendix A

Decomposition of an X_ACCP_Client_PS_SP

This appendix contains XL specifications that reflect the architecture driven decomposition of a CIM-OSA IIS X_ACCP_Client_PS_SP (X Access Protocol Client Protocol Support Service Provider) component. This is reference material for section 5.3.6.

A.1 Nodewise-distributed X_ACCP_Client_PS_Service_Agents

The following (pseudo) XL specification reflects a decomposition of the X_ACCP_Client_PS_SP which reveals its nodewise-distributed X_ACCP_Client_PS_Service_Agen

```
(* Component class: server component *)
(* Comments: a simplification of an
    X_ACCP_Client_Protocol_Support_Service_Provider *)

process X_ACCP_Client_PS_SP[X_ACCPgates,SE_ACCPgates] : noexit :=

    (* Decompose into nodewise-distributed X_ACCP_Client_PS_Service_Agents *)

         X_ACCP_Client_PS_Service_Agent[X_ACCPgate,SE_ACCPgate](1)
    ||| X_ACCP_Client_PS_Service_Agent[X_ACCPgate,SE_ACCPgate](2)
    ... ...
    ||| X_ACCP_Client_PS_Service_Agent[X_ACCPgate,SE_ACCPgate](n)

endproc (* X_ACCP_Client_PS_SP *)
```

A.2 Decomposition of an X_ACCP_Client_PS_Service_Agent

The following (pseudo) XL specification reflects a decomposition of a single X_ACCP_Client_PS_Service_Agent which reveals its composition in terms of *transformational*, *storage*, *resource-management* and *timeout components* and *combinators.*

```
(* Component class: server component *)
(* Comments: a simplification of an
    X_ACCP_Client_Protocol_Support_Service_Agent. *)
process X_ACCP_Client_PS_Service_Agent[X_ACCPgate,SE_ACCPgate]
                                        (id:Nat) : noexit :=

    (* Decompose into primary components *)

        capacity[X_ACCPgate](0)
    |[X_ACCPgate]|
        (
            (
                Xpdu_to_SEsdu[X_ACCPgate,SE_ACCPgate]
            ||| bypass[SE_ACCPgate]
            )
        |||
            SEsdu_to_Xpdu[X_ACCPgate,SE_ACCPgate]
        )
    |[SE_ACCPgate]|
        (hide timed_out in (
            resend_storage[SE_ACCPgate,timed_out]({})
        |[SE_ACCPgate,timed_out]|
            overdue_handler[SE_ACCPgate,timed_out]
        ))

 where

    (* Component class: resource management component *)
    (* Comments: Constrains the capacity of an
        X_ACCP_Client_PS_Service_Agent, i.e. the max. number
        of Xpdu Requests which the agent can handle concurrently. *)
    process capacity[X_ACCPgate](in_use:Nat) : noexit :=
            ([in_use lt max_capacity]->
                X_ACCPgate ? pdu:XpduSort [xPri(pdu) eq Request];
                capacity[X_ACCPgate](in_use+1))
        []
            X_ACCPgate ? pdu:XpduSort [xPri(pdu) eq Confirm];
            capacity[X_ACCPgate](in_use-1)
    endproc (* capacity *)

    (* Component class: transformational component *)
    (* Comments: Packages Xpdus in SEsdus. *)
    process Xpdu_to_SEsdu[X_ACCPgate,SE_ACCPgate] : noexit :=
            X_ACCPgate ? pdu:XpduSort [xPri(pdu) eq Request];
            SE_ACCPgate ?sdu:SEsduSort [sdu IsPackaged pdu];
            stop
        |||
            Xpdu_to_SEsdu[X_ACCPgate,SE_ACCPgate]
    endproc (* Xpdu_to_SEsdu *)

    (* Component class: transformational component *)
    (* Comments: Unpackages SEsdus to result in Xpdus. *)
    process SEsdu_to_Xpdu[X_ACCPgate,SE_ACCPgate] : noexit :=
            SE_ACCPgate ? sdu:SEpduSort [xPri(sdu) eq Reply];
            X_ACCPgate ?pdu:XpduSort [pdu IsUnpackaged sdu];
            stop
        |||
            SEsdu_to_Xpdu[X_ACCPgate,SE_ACCPgate]
    endproc (* SEsdu_to_Xpdu *)

    (* Component class: functional component *)
    (* Comments: Allows the resend_storage component to resend an SEsdu
        via SE_ACCPgate, without having to synchronise with the
        Xpdu_to_SEsdu component. *)
    process bypass[SE_ACCPgate] : noexit :=
        SE_ACCPgate ? sdu:SEsduSort [xPri(sdu) eq Request];
        bypass[SE_ACCPgate]
    endproc (* bypass *)
```

```
(* Component class: storage component *)
(* Comments: Temporarily stores SEsdu Requests. Resends the appropriate,
    stored SEsdu Request if a timed_out event occurs. Deletes an SEsdu
    Request from the store when the corresponding SEsdu Reply is recieved
    in time. *)
process resend_storage[SE_ACCPgate,timed_out](buffer:BufferSort) :noexit :=
        (* Store an sdu Request *)
         SE_ACCPgate ? sdu:SEsduSort [xPri(sdu) eq Request];
        resend_storage[SE_ACCPgate,timed_out](Insert(sdu,buffer))
    []
        (* Delete a stored sdu Request when a corresponding sdu Reply occurs *)
         SE_ACCPgate ? sdu:SEsduSort [(xPri(sdu) eq Reply) and
                                      (xSduId(sdu) IsInBySduId buffer)];
        resend_storage[SE_ACCPgate,timed_out](DeleteBySduId(xSduId(sdu),buffer))
    []
        (* Timeout - resend an sdu Request *)
        timed_out ! sduId:SEsduIdSort;
         SE_ACCPgate ? sdu:SEsduSort [(sdu IsIn buffer) and
                                      (xSduId(sdu) eq sduId)];
        (* Note that another copy of the sdu is saved to the buffer. This
            is because there must be one stored copy of a Request sdu
            for every outstanding reply sdu *)
        resend_storage[SE_ACCPgate,timed_out](Insert(sdu,buffer))
endproc (* resend_storage *)

(* Component class: timeout component *)
(* Comments: Generates a timed_out message if an SEsdu Reply is
    overdue. *)
process overdue_handler[SE_ACCPgate,timed_out] : noexit :=
          SE_ACCPgate ? sdu:SEsduSort [xPri(sdu) eq Request] @t1;
        (let sduId:SEsduIdSort = xSduId(sdu) in
            (* ...note the ID of the sdu. This means that this component
                doesnot have to buffer the whole sdu. Problems of storing
                sdus are localised within the resend_storage component *)
            (
                (* Reply sdu received within timeout period. Note, use
                    ASAP (maximal progress) directive to ensure that the
                    Reply sdu will occur as soon as it is firable *)
                SE_ACCPgate ? sdu:SEsduSort [(xPri(sdu) eq Reply) and
                                             (xSduId(sdu) eq sduId)]
                                            {setLE(t1+timeout_period)} ASAP;
                stop

            []
                (* Reply sdu not recieved within timeout period. Send
                    timed_out message immediately after timeout period *)
                timed_out ! sduId {setEQ(t1+timeout_period+1)};
                (* Allow a late Reply sdu to synchronise, but then ignore it *)
                SE_ACCPgate ? sdu:SEsduSort [(xPri(sdu) eq Reply) and
                                             (xSduId(sdu) eq sduId)];
                stop
            )
        )
      |||
        overdue_handler[SE_ACCPgate,timed_out]
  endproc (* overdue_handler *)

endproc (* X_ACCP_Client_PS_SP *)
```

Appendix B

XL specification of the SE_ACCP server-rôle component

This appendix contains the XL specification of the CIM-OSA IIS SE_ACCP (System-Wide Exchange Access-Protocol) server-rôle component. This is reference material for section 5.4.

(This XL specification is based on the LOTOS specification in [McC90b]. The LOTOS was checked for syntactic and semantic correctness using the SEDOS LOTOS tools [Mar89] before XL extensions where introduced.)

```
(*
 * CIM-OSA IIS SE_ACCP server-role interface component.
 * The gate se represents the SE_Service_Interface.
 *)

(* Component class: server-role interface component *)

specification SE_SERVICE [se] : noexit


(* ************************************************************************
 * Import XL Standard Library types.
 ************************************************************************ *)

library
   Boolean, NaturalNumber, Set, SetGeneratorFunctionsType
endlib


(* ************************************************************************
 * Define an SE-Callable-Function type.
 ************************************************************************ *)

type CallableFnType is Boolean, NaturalNumber
   sorts
      CallableFnSort
   opns
```

```
        SE_Initialize, SE_Inquire_Key, SE_Ask_Wait,
        SE_Attend, SE_Accept, SE_Tell, SE_Answer,
        SE_Terminate :  -> CallableFnSort
        _ eq _, _ ne _ : CallableFnSort, CallableFnSort -> Bool
        map : CallableFnSort -> Nat
    eqns
        forall c1, c2: CallableFnSort

        ofsort Nat
            map(SE_Initialize) = 0;
            map(SE_Inquire_Key) = succ(0);
            map(SE_Ask_Wait) = succ(succ(0));
            map(SE_Attend) = succ(succ(succ(0)));
            map(SE_Accept) = succ(map(SE_Attend));
            map(SE_Tell) = succ(map(SE_Accept));
            map(SE_Answer) = succ(map(SE_Tell));
            map(SE_Terminate) = succ(map(SE_Answer));

        ofsort Bool
            c1 eq c2 = map(c1) eq map(c2);
            c1 ne c2 = map(c1) ne map(c2);

endtype (* CallableFnType *)


(* ************************************************************************
 * Define a Return-Code type.
 ************************************************************************ *)

type RtnCodeType is
    sorts
        RtnCodeSort
    opns
        SE_Ok, SE_TimeOut,
        SE_InvalidKey, SE_OtherError :  -> RtnCodeSort
endtype (* RtnCodeType *)


(* ************************************************************************
 * Define a Primitive type.
 ************************************************************************ *)

type PrimitiveType is Boolean
    sorts
        PrimitiveSort
    opns
        Inp, Out :  -> PrimitiveSort
        _ eq _, _ ne _ : PrimitiveSort, PrimitiveSort -> Bool
    eqns
        forall x, y : PrimitiveSort

        ofsort Bool
            Inp eq Inp = True;
            Inp eq Out = False;
            Out eq Inp = False;
            Out eq Out = True;

            x ne y = not(x eq y);

endtype (* PrimitiveType *)


(* ************************************************************************
 * Define a type with unique identifier values (isomorphic to the natural
 * numbers).
 ************************************************************************ *)
```

```
type IdentType is Boolean
    sorts
        IdentSort
    opns
        Base : -> IdentSort
        AnotherIdent : IdentSort -> IdentSort
        _ eq _,
        _ ne _,
        _ lt _ : IdentSort, IdentSort -> Bool
    eqns
        forall i1, i2: IdentSort
            ofsort Bool
                Base eq Base = True;
                AnotherIdent(i1) eq AnotherIdent(i2) = i1 eq i2;
                Base eq AnotherIdent(i1) = False;
                AnotherIdent(i1) eq Base = False;

                i1 ne i2 = not(i1 eq i2);
                (* For SEDOS Tooset fix!... *)
                Base lt Base = False;
                Base lt AnotherIdent(i1) = True;
                AnotherIdent(i1) lt Base = False;
                AnotherIdent(i1) lt AnotherIdent(i2) = i1 lt i2;

endtype (* IdentType *)

(* ********************************************************************
 * Define an IIS-Service-Key type.
 ******************************************************************** *)

type KeyType is IdentType renamedby
    sortnames
        KeySort for IdentSort
    opnnames
        NewKey for AnotherIdent
endtype (* KeyType *)

(* ********************************************************************
 * Define an SE-PDU type.
 ******************************************************************** *)

type PduType is Boolean, NaturalNumber
    sorts
        PduSort
    opns
        ReqPdu, ResPdu, ErrorPdu : -> PduSort
        _ eq _,
        _ ne _ : PduSort, PduSort -> Bool
        map : PduSort -> Nat
    eqns
        forall p1, p2: PduSort
            ofsort Nat
                map(ReqPdu) = 0;
                map(ResPdu) = succ(0);
                map(ErrorPdu) = succ(succ(0));

            ofsort Bool
                p1 eq p2 = map(p1) eq map(p2);
                p1 ne p2 = map(p1) ne map(p2);

endtype (* PduType *)

(* ********************************************************************
 * Define an SE-Priority type.
 ********************************************************************** *)

type PriorityType is Boolean, NaturalNumber
   sorts
      PrioritySort
   opns
      Low, Medium, High :  -> PduSort
      _ eq _,
      _ ne _ : PrioritySort, PrioritySort -> Bool
      map : PrioritySort -> Nat
   eqns
      forall p1, p2: PrioritySort
         ofsort Nat
            map(Low) = 0;
            map(Medium) = succ(0);
            map(High) = succ(succ(0));

         ofsort Bool
            p1 eq p2 = map(p1) eq map(p2);
            p1 ne p2 = map(p1) ne map(p2);

endtype (* PriorityType *)

(* **********************************************************************
 * Define an SE-Message-Type type.
 ********************************************************************** *)

type TypeType is NaturalNumber renamedby
   sortnames
      TypeSort for Nat
endtype (* TypeType *)

(* **********************************************************************
 * Define an SE-Temperature type.
 ********************************************************************** *)

type TempType is Boolean
   sorts
      TempSort
   opns
      Warm, Cold :  -> TempSort
      _ eq _,
      _ ne _ : TempSort, TempSort -> Bool
   eqns
      forall t1, t2: TempSort

         ofsort Bool
            Warm eq Warm = True;
            Warm eq Cold = False;
            Cold eq Warm = False;
            Cold eq Cold = True;
            t1 ne t2 = not(t1 eq t2);

endtype (* TempType *)

(* **********************************************************************
 * Define a TimeOut type.
 ********************************************************************** *)

 (*
  * NOTE: only needed to help check this XL spec. using LOTOS tools.
  *)

type TimeType is NaturalNumber renamedby
```

```
   sortnames
      TimeSort for Nat
endtype (* TimeType *)

(* ***********************************************************************
 * Define a Node-ID type.
 *********************************************************************** *)

type NodeIDType is NaturalNumber renamedby
   sortnames
      NodeIDSort for Nat
endtype (* NodeIDType *)

(* ***********************************************************************
 * Define a Local-Name (SE-Name) type.
 *********************************************************************** *)

type LocalNameType is NaturalNumber renamedby
   sortnames
      LocalNameSort for Nat
endtype (* LocalNameType *)

(* ***********************************************************************
 * Define a Global-Name type.
 *********************************************************************** *)

type GNameType is LocalNameType, NodeIDType, Boolean
   sorts
      GNameSort
   opns
      _ eq _, _ ne _ : GNameSort, GNameSort -> Bool
      GName : LocalNameSort, NodeIDSort -> GNameSort
   eqns
      forall l1, l2: LocalNameSort, n1, n2: NodeIDSort

         ofsort Bool
            GName(l1, n1) eq GName(l2, n2) = (l1 eq l2) and (n1 eq n2);
            GName(l1, n1) ne GName(l2, n2) = (l1 ne l2) or (n1 ne n2);

endtype (* GNameType *)


(* ***********************************************************************
 * Define a Name-Key-Pair type.
 *********************************************************************** *)

type NPairType is KeyType, GNameType
   sorts
      NPairSort
   opns
      NPair : KeySort, GNameSort -> NPairSort
      _ eq _, _ ne _,
      _ lt _ : NPairSort, NPairSort -> Bool
   eqns
      forall k1, k2: KeySort, n1, n2: GNameSort

      ofsort Bool
         NPair(k1, n1) eq NPair(k2, n2) = (k1 eq k2) and (n1 eq n2);
         NPair(k1, n1) ne NPair(k2, n2) = (k1 ne k2) or (n1 ne n2);
         NPair(k1, n1) lt NPair(k2, n2) = k1 lt k2; (* For SEDOS Toolset fix! *)
endtype (* NPairType *)
```

```
(* ***********************************************************************
 * Define a formal Registration-DataBase type.
 *********************************************************************** *)

type FRegDBaseType is Set
   renamedby
      sortnames
         RegDBaseSort for Set
endtype (* FRegDBaseType *)

(* ***********************************************************************
 * Define a Registration-DataBase type.
 *********************************************************************** *)

type RegDBaseType is FRegDBaseType
   actualizedby NPairType, Boolean, NaturalNumber using
   sortnames
       NPairSort for Element
       Bool for FBool
       Nat for FNat
endtype (* RegDBaseType *)

(* ***********************************************************************
 * Define an enhanced Registration-DataBase type.
 *********************************************************************** *)

type EnhancedRegDBaseType is RegDBaseType
   opns
      RemoveByName : GNameSort, RegDBaseSort -> RegDBaseSort
      RemoveByKey : KeySort, RegDBaseSort -> RegDBaseSort
      FindKey : GNameSort, RegDBaseSort -> KeySort
   eqns
      forall n1, n2: GNameSort, k1, k2: KeySort, db: RegDBaseSort

         ofsort RegDBaseSort
            n1 eq n2 =>
                RemoveByName(n1, Insert(NPair(k2, n2), db))
                = db;
            n1 ne n2 =>
                RemoveByName(n1, Insert(NPair(k2, n2), db))
                = Insert(NPair(k2, n2), RemoveByName(n1, db));
            RemoveByName(n1, {}) = {};

            k1 eq k2 =>
                RemoveByKey(k1, Insert(NPair(k2, n2), db))
                = db;
            k1 ne k2 =>
                RemoveByKey(k1, Insert(NPair(k2, n2), db))
                = Insert(NPair(k2, n2), RemoveByKey(k1, db));
            RemoveByKey(k1, {}) = {};

         ofsort KeySort
            n1 eq n2 =>
                FindKey(n1, Insert(NPair(k2, n2), db))
                = k2;
            n1 ne n2 =>
                FindKey(n1, Insert(NPair(k2, n2), db))
                = FindKey(n1, db);
            FindKey(n1, {}) = Base;

endtype (* EnhancedRegDBaseType *)

(* ***********************************************************************
 * Define an AskWaitTriple type.
 ********************************************************************** *)

type AskWaitTripleType is KeyType, TimeType
   sorts
      AskWaitTripleSort
   opns
      AskWaitTriple : KeySort, TimeSort, TimeSort -> AskWaitSort
      _ eq _, _ ne _,
      _ lt _ : AskWaitTripleSort, AskWaitTripleSort -> Bool
   eqns
      forall k1, k2: KeySort, t1a, t1b, t2a, t2b: TimeSort

      ofsort Bool
         AskWaitTriple(k1, t1a, t1b) eq AskWaitTriple(k2, t2a, t2b)
            = (k1 eq k2) and (t1a eq t2a) and (t1b eq t2b);
         AskWaitTriple(k1, t1a, t1b) ne AskWaitTriple(k2, t2a, t2b)
            = (k1 ne k2) or (t1a ne t2a) or (t1b ne t2b);
         AskWaitTriple(k1, t1a, t1b) lt AskWaitTriple(k2, t2a, t2b)
            = k1 lt k2; (* For SEDOS Toolset fix! *)
endtype (* AskWaitTripleType *)

(* **********************************************************************
 * Define a formal Outstanding-AskWaits type.
 ********************************************************************** *)

type FOutStAskWaitsType is Set
   renamedby
      sortnames
         OutStAskWaitsSort for Set
endtype (* FOutStAskWaitsType *)

(* **********************************************************************
 * Define an Outstanding-AskWaits type.
 ********************************************************************** *)

type OutStAskWaitsType is FOutStAskWaitsType
   actualizedby AskWaitTripleType, Boolean, NaturalNumber using
   sortnames
      AskWaitTripleSort for Element
      Bool for FBool
      Nat for FNat
endtype (* OutStAskWaitsType *)


(* **********************************************************************
 * Define a Request-Packet type.
 ********************************************************************** *)

type ReqPktType is KeyType, PduType, TypeType
   sorts
      ReqPktSort
   opns
      ReqPkt : KeySort, KeySort, PduSort, TypeSort -> ReqPktSort
      _ eq _,
      _ ne _ ,
      _ lt _ : ReqPktSort, ReqPktSort -> Bool
      xInitiatorKey,
      xTargetKey : ReqPktSort -> KeySort
      xType : ReqPktSort -> TypeSort
      xPdu : ReqPktSort -> PduSort
      _ TypeEqs _ : TypeSort, ReqPktSort -> Bool
      _ ReciEqs _ : KeySort, ReqPktSort -> Bool
   eqns
```

```
    forall kreq1, kreq2, kres1, kres2: KeySort,
           pdu1, pdu2: PduSort, ty1, ty2: TypeSort

        ofsort Bool
            ty1 TypeEqs ReqPkt(kreq2, kres2, pdu2, ty2)
                = ty1 eq ty2;

            kres1 ResEqs ReqPkt(kreq2, kres2, pdu2, ty2)
                = kres1 eq kres2;

            ReqPkt(kreq1, kres1, pdu1, ty1) eq
                    ReqPkt(kreq2, kres2, pdu2, ty2)
                = (kreq1 eq kreq2) and (kres1 eq kres2) and
                  (pdu1 eq pdu2) and (ty1 eq ty2);
            ReqPkt(kreq1, kres1, pdu1, ty1) ne
                    ReqPkt(kreq2, kres2, pdu2, ty2)
                = (kreq1 ne kreq2) or (kres1 ne kres2) or
                  (pdu1 ne pdu2) or (ty1 ne ty2);
            ReqPkt(kreq1, kres1, pdu1, ty1) lt
                    ReqPkt(kreq2, kres2, pdu2, ty2)
                = kreq1 ne kreq2; (* SEDOS Toolset fix! *)

        ofsort KeySort
            xInitiatorKey(ReqPkt(kreq1, kres1, pdu1, ty1)) = kreq1;
            xTargetKey(ReqPkt(kreq1, kres1, pdu1, ty1)) = kres1;

        ofsort TypeSort
            xType(ReqPkt(kreq1, kres1, pdu1, ty1)) = ty1;

        ofsort PduSort
            xPdu(ReqPkt(kreq1, kres1, pdu1, ty1)) = pdu1;

endtype (* ReqPktType *)


(* ********************************************************************
 * Define a formal Requests-Set type.
 ******************************************************************** *)

type FReqsSetType is Set
   renamedby
      sortnames
         ReqsSetSort for Set
endtype (* FReqsSetType *)


(* ********************************************************************
 * Define a Requests-Set type.
 ******************************************************************** *)

type ReqsSetType is FReqsSetType
   actualizedby ReqPktType, Boolean, NaturalNumber using
   sortnames
      ReqPktSort for Element
      Bool for FBool
      Nat for FNat
endtype (* ReqsSetType *)


(* ********************************************************************
 * Define an enhanced Requests-Set type.
 ******************************************************************** *)

type EnhancedReqSetType is ReqsSetType, ReqPktType, TPairType
   opns
      KeyAndTypeIn : KeySort, TypeSort, ReqsSetSort -> Bool
    RtnAType : TPairSort, ReqsSetSort -> TypeSort
  eqns
    forall rset: ReqsSetSort, r1: ReqPktSort,
           k: KeySort, ty: TypeSort

      ofsort Bool
        (k ReciEqs r1) and (ty TypeEqs r1) =>
            KeyAndTypeIn(k, ty, Insert(r1, rset)) = True;
        not((k ReciEqs r1) and (ty TypeEqs r1)) =>
            KeyAndTypeIn(k, ty, Insert(r1, rset))
            = KeyAndTypeIn(k, ty, rset);
        KeyAndTypeIn(k, ty, {}) = False;

      ofsort TypeSort
        KeyAndTypeIn(k, ty, rset) =>
            RtnAType(TPair(k, ty), rset) = ty;
        not(KeyAndTypeIn(k, ty, rset)) =>
            RtnAType(TPair(k, ty), rset) = 0;

endtype (* EnhancedReqSetType *)


(* *********************************************************************
 * Define a Responses-Packet type.
 ********************************************************************* *)

type ResPktType is KeyType, PduType, TimeType
  sorts
    ResPktSort
  opns
    ResPkt : KeySort, PduSort, TimeSort -> ResPktSort
    _ eq _, _ ne _,
    _ lt _ : ResPktSort, ResPktSort -> Bool
  eqns
    forall kreq1, kreq2: KeySort, pdu1, pdu2: PduSort, t1, t2: TimeSort

      ofsort Bool
        ResPkt(kreq1, pdu1, t1) eq ResPkt(kreq2, pdu2, t2)
            = (kreq1 eq kreq2) and (pdu1 eq pdu2) and (t1 eq t2);
        ResPkt(kreq1, pdu1) ne ResPkt(kreq2, pdu2)
            = (kreq1 ne kreq2) or (pdu1 ne pdu2) or (t1 ne t2);
        ResPkt(kreq1, pdu1) lt ResPkt(kreq2, pdu2)
            = kreq1 lt kreq2; (* For SEDOS Toolset fix! *)

endtype (* ResPktType *)

(* *********************************************************************
 * Define a formal Responses-Set type.
 ********************************************************************* *)

type FResSetType is Set
  renamedby
    sortnames
      ResSetSort for Set
endtype (* FResSetType *)

(* *********************************************************************
 * Define a Responses-Set type.
 ********************************************************************* *)

type ResSetType is FResSetType
  actualizedby ResPktType, Boolean, NaturalNumber using
  sortnames
    ResPktSort for Element
```

```
        Bool for FBool
        Nat for FNat
endtype (* ResSetType *)

(* **********************************************************************
 * Define an enhanced Responses-Set type.
 ********************************************************************** *)

type EnhancedResSetType is ResSetType
    opns
        RemoveByKey : KeySort, ResSetSort -> ResSetSort
    eqns
        forall k1, k2: KeySort, pdu: PduSort, t1: TimeSort, rs: ResSetSort

        ofsort ResSetSort
            k1 eq k2 =>
                RemoveByKey(k1, Insert(ResPkt(k2, pdu, t1), rs))
                = RemoveByKey(k1, rs);
            k1 ne k2 =>
                RemoveByKey(k1, Insert(ResPkt(k2, pdu, t1), rs))
                = Insert(ResPkt(k2, pdu, t1), RemoveByKey(k1, rs));
            RemoveByKey(k1, {}) = {};

endtype (* EnhancedResSetType *)

(* **********************************************************************
 * Define a Type-Key-Pair type.
 ********************************************************************** *)

type TPairType is KeyType, TypeType
    sorts
        TPairSort
    opns
        TPair : KeySort, TypeSort -> TPairSort
        _ eq _, _ ne _,
        _ lt _ : TPairSort, TPairSort -> Bool
    eqns
        forall k1, k2: KeySort, t1, t2: TypeSort

        ofsort Bool
            TPair(k1, t1) eq TPair(k2, t2) = (k1 eq k2) and (t1 eq t2);
            TPair(k1, t1) ne TPair(k2, t2) = (k1 ne k2) or (t1 ne t2);
            TPair(k1, t1) lt TPair(k2, t2) = k1 lt k2; (* For SEDOS Toolset fix! *)
endtype (* TPairType *)

(* **********************************************************************
 * Define a formal Outstanding-Inquires type.
 ********************************************************************** *)

type FOutStInquiresType is Set
    renamedby
        sortnames
            OutStInquiresSort for Set
endtype (* FOutStInquiresType *)

(* **********************************************************************
 * Define an Outstanding-Inquires type.
 ********************************************************************** *)

type OutStInquiresType is FOutStInquiresType
    actualizedby NPairType, Boolean, NaturalNumber using
    sortnames
        NPairSort for Element
        Bool for FBool
        Nat for FNat
endtype (* OutStInquiresType *)


(* ***********************************************************************
 * Define an enhanced Outstanding-Inquires type.
 *********************************************************************** *)

type EnhancedOutStInquiresType is OutStInquiresType
    opns
        RtnInqNameByKey : KeySort, OutStInquiresSort -> GNameSort
        RemoveByKey : KeySort, OutStInquiresSort -> OutStInquiresSort
    eqns
        forall k1, k2: KeySort, na: GNameSort, oi: OutStInquiresSort

        ofsort GNameSort
            k1 eq k2 =>
                RtnInqNameByKey(k1, Insert(NPair(k2, na), oi))
                = na;
            k1 ne k2 =>
                RtnInqNameByKey(k1, Insert(NPair(k2, na), oi))
                = RtnInqNameByKey(k1, oi) ;
            RtnInqNameByKey(k1, {} of OutStInquiresSort) = GName(0, 0);

        ofsort OutStInquiresSort
            k1 eq k2 =>
                RemoveByKey(k1, Insert(NPair(k2, na), oi))
                = RemoveByKey(k1, oi);
            k1 ne k2 =>
                RemoveByKey(k1, Insert(NPair(k2, na), oi))
                = Insert(NPair(k2, na), RemoveByKey(k1, oi));
            RemoveByKey(k1, {}) = {};

endtype (* EnhancedOutStInquiresType *)


(* ***********************************************************************
 * Define a formal Outstanding-Attends type.
 *********************************************************************** *)

type FOutStAttendsType is Set
    renamedby
        sortnames
            OutStAttendsSort for Set
endtype (* FOutStAttendsType *)


(* ***********************************************************************
 * Define an Outstanding-Attends type.
 *********************************************************************** *)

type OutStAttendsType is FOutStAttendsType
    actualizedby TPairType, Boolean, NaturalNumber using
    sortnames
        TPairSort for Element
        Bool for FBool
        Nat for FNat
endtype (* OutStAttendsType *)


(* ***********************************************************************
 * Define an enhanced Outstanding-Attends type.
 *********************************************************************** *)
```

```
type EnhancedOutStAttendsType is OutStAttendsType
   opns
      xTPairByKey : KeySort, OutStAttendsSort -> TPairSort
   eqns
      forall k1, k2: KeySort, t1, t2: TypeSort, oa: OutStAttendsSort

      ofsort TPairSort
         k1 eq k2 =>
            xTPairByKey(k1, Insert(TPair(k2, t2), oa))
            = TPair(k2, t2);
         k1 ne k2 =>
            xTPairByKey(k1, Insert(TPair(k2, t2), oa))
            = xTPairByKey(k1, oa);
         xTPairByKey(k1, {}) = TPair(Base, 0);

endtype (* EnhancedOutStAttendsType *)

(* ***********************************************************************
 * Define a formal Outstanding-Accepts type.
 *********************************************************************** *)

type FOutStAcceptsType is Set
   renamedby
      sortnames
         OutStAcceptsSort for Set
endtype (* FOutStAcceptsType *)

(* ***********************************************************************
 * Define an Outstanding-Accepts type.
 *********************************************************************** *)

type OutStAcceptsType is FOutStAcceptsType
   actualizedby TPairType, Boolean, NaturalNumber using
   sortnames
      TPairSort for Element
      Bool for FBool
      Nat for FNat
endtype (* OutStAcceptsType *)


(* ***********************************************************************
 * Define an enhanced Outstanding-Accepts type.
 *********************************************************************** *)

type EnhancedOutStAcceptsType is OutStAcceptsType
   opns
      RemoveByKey : KeySort, OutStAcceptsSort -> OutStAcceptsSort
   eqns
      forall k1, k2: KeySort, ty: TypeSort, oa: OutStAcceptsSort

      ofsort OutStAcceptsSort
         k1 eq k2 =>
            RemoveByKey(k1, Insert(TPair(k2, ty), oa))
            = RemoveByKey(k1, oa);
         k1 ne k2 =>
            RemoveByKey(k1, Insert(TPair(k2, ty), oa))
            = Insert(TPair(k2, ty), RemoveByKey(k1, oa));
         RemoveByKey(k1, {}) = {};

endtype (* EnhancedOutStAcceptsType *)

(* ***********************************************************************
 * Define an SE-SDU type.
****************************************************************** *)

type SeSduType is CallableFnType, PrimitiveType, PriorityType, KeyType,
                  GNameType, PduType, TempType, TypeType, TimeType
                  RtnType
  sorts
     SeSduSort
  opns
     ItIn : GNameSort, TempSort, PrioritySort
              -> SeSduSort (* SE_Initialise Inp *)
     ItOu : GNameSort, KeySort, RtnCodeSort
              -> SeSduSort (* SE_Initialise Out *)
     IqIn : KeySort, GNameSort, PrioritySort
              -> SeSduSort (* SE_Inquire_Key Inp *)
     IqOu : KeySort, KeySort, RtnCodeSort
              -> SeSduSort (* SE_Inquire_Key Out *)
     AwIn : KeySort, KeySort, PduSort, TypeSort, TimeSort, PrioritySort
              -> SeSduSort (* SE_Ask_Wait Inp *)
     AwOu : KeySort, PduSort, RtnCodeSort
              -> SeSduSort (* SE_Ask_Wait Out *)
     AtIn : KeySort, TypeSort, PrioritySort
              -> SeSduSort (* SE_Attend Inp *)
     AtOu : KeySort, TypeSort, RtnCodeSort
              -> SeSduSort (* SE_Attend Out *)
     AcIn : KeySort, TypeSort, PrioritySort
              -> SeSduSort (* SE_Accept Inp *)
     AcOu : KeySort, KeySort, PduSort, RtnCodeSort
              -> SeSduSort (* SE_Accept Out *)
     AnIn : KeySort, KeySort, PduSort, PrioritySort
              -> SeSDuSort (* SE_Answer Inp *)
     AnOu : KeySort, RtnCodeSort
              -> SeSDuSort (* SE_Answer Out *)
     TmIn : KeySort, TempSort, PrioritySort
              -> SeSduSort (* SE_Terminate Inp *)
     TmOu : KeySort, RtnCodeSort
              -> SeSduSort (* SE_Terminate Out *)

     xFn : SeSduSort -> CallableFnSort
     xPri : SeSduSort -> PrimitiveSort
     xName : SeSduSort -> GNameSort
     xPriority : SeSduSort -> PrioritySort
     xCallerKey,
     xNKey, xQKey,
     xInitiatorKey,
     xCalledKey,
     xTargetKey : SeSduSort -> KeySort
     xTemp : SeSduSort -> TempSort
     xType : SeSduSort -> TypeSort
     xTimeout : SeSduSort -> TimeSort
     xPdu : SeSduSort -> PduSort
     xRtnCode : SeSduSort -> RtnCodeSort


   eqns
     forall k1, k2: KeySort, temp: TempSort, ty: TypeSort, na: GNameSort,
            pr: PrioritySort, pdu: PduSort, ti: TimeSort, rt: RtnCodeSort

     ofsort CallableFnSort
        xFn(ItIn(na, temp, pr)) = SE_Initialize;
        xFn(ItOu(na, k1, rt)) = SE_Initialize;
        xFn(IqIn(k1, na, pr)) = SE_Inquire_Key;
        xFn(IqOu(k1, k2, rt)) = SE_Inquire_Key;
        xFn(AwIn(k1, k2, pdu, ty, ti, pr)) = SE_Ask_Wait;
        xFn(AwOu(k1, pdu, rt)) = SE_Ask_Wait;
        xFn(AtIn(k1, ty, pr)) = SE_Attend;
        xFn(AtOu(k1, ty, rt)) = SE_Attend;
```

```
    xFn(AcIn(k1, ty, pr)) = SE_Accept;
    xFn(AcOu(k1, k2, pdu, rt)) = SE_Accept;
    xFn(AnIn(k1, k2, pdu, pr)) = SE_Answer;
    xFn(AnOu(k1, rt)) = SE_Answer;
    xFn(TmIn(k1, temp, pr)) = SE_Terminate;
    xFn(TmOu(k1, rt)) = SE_Terminate;

ofsort PrimitiveSort
    xPri(ItIn(na, temp, pr)) = Inp;
    xPri(ItOu(na, k1, rt)) = Out;
    xPri(IqIn(k1, na, pr)) = Inp;
    xPri(IqOu(k1, k2, rt)) = Out;
    xPri(AwIn(k1, k2, pdu, ty, ti, pr)) = Inp;
    xPri(AwOu(k1, pdu, rt)) = Out;
    xPri(AtIn(k1, ty, pr)) = Inp;
    xPri(AtOu(k1, ty, rt)) = Out;
    xPri(AcIn(k1, ty, pr)) = Inp;
    xPri(AcOu(k1, k2, pdu, rt)) = Out;
    xPri(AnIn(k1, k2, pdu, pr)) = Inp;
    xPri(AnOu(k1, rt)) = Out;
    xPri(TmIn(k1, temp, pr)) = Inp;
    xPri(TmOu(k1, rt)) = Out;

ofsort PrioritySort
    xPriority(ItIn(na, temp, pr)) = pr;
    xPriority(IqIn(k1, na, pr)) = pr;
    xPriority(AwIn(k1, k2, pdu, ty, ti, pr)) = pr;
    xPriority(AtIn(k1, ty, pr)) = pr;
    xPriority(AcIn(k1, ty, pr)) = pr;
    xPriority(AnIn(k1, k2, pr)) = pr;
    xPriority(TmIn(k1, temp, pr)) = pr;

ofsort GNameSort
    xName(ItIn(na, temp, pr)) = na;
    xName(ItOu(na, k1, rt)) = na;
    xName(IqIn(k1, na, pr)) = na;

ofsort KeySort
    xNKey(ItOu(na, k1, rt)) = k1;
    xCallerKey(IqIn(k1, na, pr)) = k1;
    xCallerKey(IqOu(k1, k2, rt)) = k1;
    xQKey(IqOu(k1, k2, rt)) = k2;
    xCallerKey(AwIn(k1, k2, pdu, ty, ti, pr)) = k1;
    xCalledKey(AwIn(k1, k2, pdu, ty, ti, pr)) = k2;
    xCallerKey(AwOu(k1, pdu, rt)) = k1;
    xCallerKey(AtIn(k1, ty, pr)) = k1;
    xCallerKey(AtOu(k1, ty, rt)) = k1;
    xCallerKey(AcIn(k1, ty, pr)) = k1;
    xCallerKey(AcOu(k1, k2, pdu, rt)) = k1;
    xInitiatorKey(AcOu(k1, k2, pdu, rt)) = k2;
    xCallerKey(AnIn(k1, k2, pdu, pr)) = k1;
    xTargetKey(AnIn(k1, k2, pdu, pr)) = k2;
    xCallerKey(AnOu(k1, rt)) = k1;
    xCallerKey(TmIn(k1, temp, pr)) = k1;
    xCallerKey(TmOu(k1, rt)) = k1;

ofsort TempSort
    xTemp(ItIn(na, temp, pr)) = temp;
    xTemp(TmIn(k1, temp, pr)) = temp;

ofsort TypeSort

    xType(AtIn(k1, ty, pr)) = ty;
    xType(AtOu(k1, ty, rt)) = ty;
    xType(AcIn(k1, ty, pr)) = ty;

ofsort PduSort
```

```
        xPdu(AwIn(k1, k2, pdu, ty, ti, pr)) = pdu;
        xPdu(AwOu(k1, pdu, rt)) = pdu;
        xPdu(AcOu(k1, k2, pdu, rt)) = pdu;
        xPdu(AnIn(k1, k2, pdu, pr)) = pdu;

     ofsort TimeSort

        xTimeout(AwIn(k1, k2, pdu, ty, ti, pr)) = ti;

endtype (* SeSduType *)




(* ***********************************************************************
 * Comments: Highest level behavioural expression.
 * Events occurring at the gate se model the allowable behaviour and
 * information exchange between the SE_Service and the SE_Service-Users at
 * the SE_Service-Interface. PERFORMANCE and FUNCTIONALITY appropriately
 * constrain events occurring at the gate se.
 *********************************************************************** *)

behaviour

     PERFORMANCE[se]
   ||
     FUNCTIONALITY[se]

where

   (* ********************************************************************
    * Component class: performance component *)
    * Comments: Embodies performance concerned constraints.
    ******************************************************************** *)
   process PERFORMANCE[se] : noexit :=

         PROB_SRV_TIME[se]
      ||
         PRIORITY_SELECTION[se]
      ||
         CAPACITY[se](0)

   where

      (* *****************************************************************
       * Component class: probability/stopwatch component
       * Comments: If the function call is not a waiting function
       * (i.e. an SE_Ask_wait or SE_Attend_Wait) then constrain the actual
       * service_time of the call by: 1) noting the time at which the Input
       * event OCCURS, and then 2) specifying earliest time at which the
       * Output event is OFFERED. The actual service_time equals the 'Output
       * OFFER time' minus the 'Input OCCURRENCE time'. The actual service
       * time is constrained to be either: 1) <= target_srv_time with a
       * probability of 0.999, or 2) > target_srv_time with a probability
       * of 0.001.
       *
       * (If the function call is a waiting function (e.g.. SE_Ask_Wait) then
       * the quantatitative timing constraints are specified with the functional
       * constraints (e.g. in O_ASK_WAIT).)
       *
       * Also, this component enforces the pairing of an Input
       * event with its complemenary Output event. It does by ensuring that
       * both the Input event and Output event in a pair contain the same
       * Caller Key or Name.
       *
```

```
* CONSTANT: target_srv_time should be a term ofsort TimeSort.
****************************************************************** *)

process PROB_SRV_TIME[se] : noexit :=

   (
     (* For each Input-Output event pairing do... *)
     se ? sdu1: SeSduSort [xPri(sdu1) eq Inp] @t1;
                  (* ...Accept any Input event and note the time in
                        the variable t1 *)
     (

       (*-----------------------------------------------------------*)
       [(xFn(sdu1) ne SE_Ask_Wait)] -> (* Not a waiting function *)
       (
         (*
          * Choose an 'actual service time' value...
          *)
         (
           (* probability of 0.999 of taking this branch
            * where we restrict
            * 'actual service time' <= 'target service time'
            *)
           choice act_srv_time:TimeSort []
               [act_srv_time le target_srv_time] ->
                   exit(act_srv_time)
         [=0.999]
           (* probability of 0.001 of taking this branch
            * where we restrict
            * 'actual service time' > 'target service time'
            *)
           choice act_srv_time:TimeSort []
               [act_srv_time gt target_srv_time] ->
                   exit(act_srv_time)
         )
         >> accept act_srv_time:TimeSort in


         (*
          * Now use this act_srv_time value to approriately
          * constrain the Output event...
          *)
         (
             [xFn(sdu1) eq SE_Initialize] ->
                 (* The Input and Output event pair for
                  * SE_Initialize must contain the same name...
                  *)
                 se ? sdu2: SeSduSort [(xPri(sdu2) eq Out) and
                                      (xName(sdu1) eq xName(sdu2))]
                                      {setGE(t1+act_srv_time)};
                 stop
          []
             [not(xFn(sdu1) eq SE_Initialize)] ->
                 (* The Input and Output event pair for
                  * other functions must contain the same key...
                  *)
                 se ? sdu2: SeSduSort [(xPri(sdu2) eq Out) and
                                      (xCallerKey(sdu1)
                                           eq xCallerKey(sdu2))]
                                     {setGE(t1+act_srv_time)};
                 stop
         )
       )
       (*--------------------------------*)
     []
       (*-----------------------------------------------------------*)
```

```
            [(xFn(sdu1) eq SE_Ask_Wait)] -> (* A waiting function *)
                                            (* Note that SE_Attend_Wait
                                               has not been 'implemented' *)
              (* We leave the responsibity for imposing quantitative
               * timing constraints upon SE_Ask_Wait invocations to the
               * O_ASK_WAIT component because, unlike for the other
               * SE-Function-Calls, SE_Ask_Wait timing constraints
               * are much more involved with the functional
               * constraints.
               *)
              se ? sdu2: SeSduSort [(xPri(sdu2) eq Out) and
                                    (xCallerKey(sdu1) eq xCallerKey(sdu2))];
              stop
           (*-------------------------------------------------*)
        )
     )
    |||
    (* Do other Input-Output event pairings... *)
    PROB_SRV_TIME[se]

endproc (* PROB_SRV_TIME *)


(* ****************************************************************
 * Component class: priority component
 * Comments: PRIORITY_SELECTION influences the order of occurrence of
 * SE-Function-Call Inputs, based on their priority parameter. (We
 * say "influences" because other IIS entities may impose additional
 * influencing priority constraints, and the resulting conjunction
 * may not exactly reflect the priority ordering wishes of the
 * SE_Service.) No priority ordering is defined amongst Output events,
 * or between Output and Input events.
 *
 * CONSTANTS: High, Medium and Low ofsort PrioritySort have been
 * assigned the Nat values 9, 6 and 3. These values represent the
 * XL priority values. All prioritized events are in priority
 * class 0.
 **************************************************************** *)

process PRIORITY_SELECTION[se]
      se ? sdu1: SeSduSort [(xPri(sdu1) eq Inp) and
                            (xPriority(sdu1) eq High)]
                            #(0,High);
      PRIORITY_SELECTION[se]
    []
      se ? sdu1: SeSduSort [(xPri(sdu1) eq Inp) and
                            (xPriority(sdu1) eq Medium)]
                            #(0,Medium);
      PRIORITY_SELECTION[se]
    []
      se ? sdu1: SeSduSort [(xPri(sdu1) eq Inp) and
                            (xPriority(sdu1) eq Low)]
                            #(0,Low);
      PRIORITY_SELECTION[se]
    []
      se ? sdu2: SeSduSort [xPri(sdu2) eq Out];
      PRIORITY_SELECTION[se]

endproc (* PRIORITY_SELECTION *)


(* ****************************************************************
 * Component class: resource management component
 * Comments: Enforces an upper limit (max_capacity) on the number of
 * function calls concurrently handled by the SE_Service.
 *
 * CONSTANT: max_capacity should be a term ofsort Nat.
```

```
  ****************************************************************** *)
  process CAPACITY[se](curr_num:Nat)

      [curr_num lt max_capacity]->
          (* The maximum capacity of SE_Service has not yet been reached
           * or exceeded so let another SE_Function_CALL invocation
           * occur...
           *)
          se ? sdu1: SeSduSort [xPri(sdu1) eq Inp];
          CAPACITY[se](curr_num+1)
     []
      se ? sdu2: SeSduSort [xPri(sdu1) eq Out];
      (* An SE_Function_Call Output event means that the SE_Service is
       * no longer processing this call, so decrement the curr_num count
       *)
      CAPACITY[se](curr_num-1)

  endproc (* CAPACITY *)


endproc (* PERFORMANCE *)

(* ***********************************************************************
 * Component class: functional component
 * Comments: Embodies primarily functional concerned constraints.
 ********************************************************************** *)
process FUNCTIONALITY[se] : noexit :=

   IO[se]({} of ReqsSetSort,
        {} of ResSetSort,
        {} of OutStAskWaitsSort,
        {} of RegDBaseSort,
        {} of OutStInquiresSort,
        {} of OutStAttendsSort,
        {} of OutStAcceptsSort,
        Base of KeySort)

 where

   (* ********************************************************************
    * Component class: transformational component
    * Comments: Given that the previous history of event occurrences at
    * the SE server-role interface (se) will affect the information
    * content and behaviour of all subsequent events occurring at se:
    * IO will either allow an Input event to happen and update the
    * "state information" accordingly; or will allow a suitable Output
    * event to happen given the current "state information" (which
    * captures all important aspects of the previous history of event
    * occurrences at se).
    ****************************************************************** *)

    process IO [se] (reqset: ReqsSetSort,
                     resset: ResSetSort,
                     outstaskwaits: OutStAskWaitsSort,
                     regset: RegDBaseSort,
                     outstinquires: OutStInquiresSort,
                     outstattends: OutStAttendsSort,
                     outstaccepts: OutStAcceptsSort,
                     generator: KeySort)
    : noexit :=

        (
            (* An Input event occurs... *)
            INPUT[se] (reqset, resset, outstaskwaits, regset, outstinquires,
                       outstattends, outstaccepts, generator)
         []
    (* An Output event occurs... *)
    OUTPUT[se] (reqset, resset, outstaskwaits, regset, outstinquires,
                 outstattends, outstaccepts, generator)
) >> accept (* the new "state" information *)
        nreqset: ReqsSetSort,
        nresset: ResSetSort,
        noutstaskwaits: OutStAskWaitsSort,
        nregset: RegDBaseSort,
        noutstinquires: OutStInquiresSort,
        noutstattends: OutStAttendsSort,
        noutstaccepts: OutStAcceptsSort,
        ngenerator: KeySort
in
     IO[se](nreqset, nresset, noutstaskwaits, nregset,
            noutstinquires, noutstattends,
            noutstaccepts, ngenerator) (* ... and recurse *)

where

  (* *****************************************************************
   * Component class: functional component
   * Comments: INPUT deals with primarily functional constraints
   * over events which represent Input SE SDUs.
   ***************************************************************** *)

  process INPUT [se] (reqset: ReqsSetSort,
                resset: ResSetSort,
                outstaskwaits: OutStAskWaitsSort,
                regset: RegDBaseSort,
                outstinquires: OutStInquiresSort,
                outstattends: OutStAttendsSort,
                outstaccepts: OutStAcceptsSort,
                generator: KeySort)
   exit (ReqsSetSort, ResSetSort, OutStAskWaitsSort, RegDBaseSort,
         OutStInquiresSort, OutStAttendsSort,
         OutStAcceptsSort, KeySort) :=

     (* Separate LOTOS process templates to impose constraints on
      * and deal with the processing for each type of
      * SE-Callable-Function Input event.
      *)

     [
       I_INITIALIZE[se](reqset, resset, outstaskwaits, regset,
                              outstinquires, outstattends,
                              outstaccepts, generator)
     [] I_INQUIRE_KEY[se](reqset, resset, outstaskwaits, regset,
                              outstinquires, outstattends,
                              outstaccepts, generator)
     [] I_ASK_WAIT[se](reqset, resset, outstaskwaits, regset,
                          outstinquires, outstattends,
                          outstaccepts, generator)
     [] I_ATTEND[se](reqset, resset, outstaskwaits, regset,
                       outstinquires, outstattends,
                       outstaccepts, generator)
     [] I_ACCEPT[se](reqset, resset, outstaskwaits, regset,
                       outstinquires, outstattends,
                       outstaccepts, generator)
     [] I_ANSWER[se](reqset, resset, outstaskwaits, regset,
                       outstinquires, outstattends,
                       outstaccepts, generator)
     [] I_TELL[se](reqset, resset, outstaskwaits, regset,
                    outstinquires, outstattends,
                    outstaccepts, generator)
     [] I_TERMINATE[se](reqset, resset, outstaskwaits, regset,
                          outstinquires, outstattends,
                          outstaccepts, generator)
    )

where

    (* *********************************************************
     * Component class: storage component
     * Comments: Initializes an SE_Service_user with the
     * registration database (regset).
     ********************************************************* *)

     process I_INITIALIZE [se] (reqset: ReqsSetSort,
                                resset: ResSetSort,
                                outstaskwaits: OutStAskWaitsSort,
                                regset: RegDBaseSort,
                                outstinquires: OutStInquiresSort,
                                outstattends: OutStAttendsSort,
                                outstaccepts: OutStAcceptsSort,
                                generator: KeySort)
     : exit (ReqsSetSort, ResSetSort, OutStAskWaitsSort, RegDBaseSort,
         OutStInquiresSort, OutStAttendsSort,
         OutStAcceptsSort, KeySort) :=

         se ? sdu: SeSduSort [(xPri(sdu) eq Inp) and
                              (xFn(sdu) eq SE_Initialize)];
         ([xTemp(sdu) eq Cold] -> (* Cold start *)
             exit (reqset,
                   RemoveByKey(FindKey(xName(sdu), regset), resset),
                   outstaskwaits,
                   Insert(NPair(NewKey(generator), xName(sdu)),
                          RemoveByName(xName(sdu), regset)),
                   outstinquires, outstattends, outstaccepts,
                   NewKey(generator))
                   (* ...Remove all old response message packets which
                         still exist for this name *)
                   (* ...Remove any old registration entry for this
                         entity and create a new one *)
                   (* ...Rtn new generator key value *)
          []
           [xTemp(sdu) eq Warm] -> (* Warm start *)
             exit (reqset, resset, outstaskwaits, regset,
                   outstinquires, outstattends, outstaccepts,
                   generator)
                   (* ...Use old registration entry *)
                   (* QUESTION TO FRB: What if such an entry does
                      not exist? *)
         )

      endproc (* I_INITALIZE *)


     (* *********************************************************
      * Component class: functional component
      * Comments: Constrains and processes the Input event occurrance
      * for SE_Inquire_Key.
      ********************************************************* *)

     process I_INQUIRE_KEY [se] (reqset: ReqsSetSort,
                                 resset: ResSetSort,
                                 outstaskwaits: OutStAskWaitsSort,
                                 regset: RegDBaseSort,
                                 outstinquires: OutStInquiresSort,
                                 outstattends: OutStAttendsSort,
                                 outstaccepts: OutStAcceptsSort,
                                 generator: KeySort)
     : exit (ReqsSetSort, ResSetSort, OutStAskWaitsSort, RegDBaseSort,
         OutStInquiresSort, OutStAttendsSort,
    OutStAcceptsSort, KeySort) :=

  se ? sdu: SeSduSort [(xPri(sdu) eq Inp) and
                       (xFn(sdu) eq SE_Inquire_Key)];
  exit (reqset, resset, outstaskwaits, regset,
        Insert(NPair(xCallerKey(sdu), xName(sdu)), outstinquires),
        outstattends, outstaccepts, generator)
        (* ...Note the outstanding SE_Inquire *)

endproc (* I_INQUIRE_KEY *)

(* ************************************************************
 * Component class: functional component/stopwatch component
 * Comments: This component specifies both the functional and
 * stopwatch aspects of the SE_Ask_Wait Input event.
 * We note the occurrence time of the Input event, for this
 * time value is needed by O_ASK_WAIT to calculate when the
 * SE_Ask_Wait invocation times out.
 ************************************************************ *)

process I_ASK_WAIT [se] (reqset: ReqsSetSort,
                         resset: ResSetSort,
                         outstaskwaits: OutStAskWaitsSort,
                         regset: RegDBaseSort,
                         outstinquires: OutStInquiresSort,
                         outstattends: OutStAttendsSort,
                         outstaccepts: OutStAcceptsSort,
                         generator: KeySort)
 exit (ReqsSetSort, ResSetSort, OutStAskWaitsSort, RegDBaseSort,
    OutStInquiresSort, OutStAttendsSort,
    OutStAcceptsSort, KeySort) :=

  se ? sdu: SeSduSort [(xPri(sdu) eq Inp) and
                       (xFn(sdu) eq SE_ASk_Wait)]
                      @invoke_time;
        (* ...Note the occurrence time of the SE_Ask_Wait
              Input event --- this time is needed later to
              to ascertain if this SE_Ask_Wait times-out *)
  exit (Insert(ReqPkt(xCallerKey(sdu), xCalledKey(sdu),
              xPdu(sdu), xType(sdu)), reqset),
        resset,
        Insert(AskWaitTriple(xCallerKey(sdu), invoke_time,
                                                   xTimeout(sdu)),
              outstaskwaits),
        regset, outstinquires, outstattends,
        outstaccepts, generator)
        (* ...Create a request packet *)
        (* ...Note the outstanding SE_Ask_Wait *)

endproc (* I_ASK_WAIT *)

(* ************************************************************
 * Component class: functional component
 * Comments: Constrains and processes the Input event occurrence
 * for SE_Attend.
 ************************************************************ *)

process I_ATTEND [se] (reqset: ReqsSetSort,
                       resset: ResSetSort,
                       outstaskwaits: OutStAskWaitsSort,
                       regset: RegDBaseSort,
                       outstinquires: OutStInquiresSort,
                       outstattends: OutStAttendsSort,
                       outstaccepts: OutStAcceptsSort,
                       generator: KeySort)
```

```
: exit (ReqsSetSort, ResSetSort, OutStAskWaitsSort, RegDBaseSort,
     OutStInquiresSort, OutStAttendsSort,
     OutStAcceptsSort, KeySort) :=

    se ? sdu: SeSduSort [(xPri(sdu) eq Inp) and
                                   (xFn(sdu) eq SE_Attend)];
    exit (reqset, resset, outstaskwaits,
            regset, outstinquires,
           Insert(TPair(xCallerKey(sdu), xType(sdu)), outstattends),
           outstaccepts, generator)
           (* ...Note the outstanding SE_Attend *)

endproc (* I_ATTEND *)


(* ***********************************************************
 * Component class: functional component
 * Comments: Constrains and processes the Input event occurrence
 * for SE_Accept.
 *********************************************************** *)

process I_ACCEPT [se] (reqset: ReqsSetSort,
                               resset: ResSetSort,
                               outstaskwaits: OutStAskWaitsSort,
                               regset: RegDBaseSort,
                               outstinquires: OutStInquiresSort,
                               outstattends: OutStAttendsSort,
                               outstaccepts: OutStAcceptsSort,
                               generator: KeySort)
: exit (ReqsSetSort, ResSetSort, OutStAskWaitsSort, RegDBaseSort,
     OutStInquiresSort, OutStAttendsSort,
     OutStAcceptsSort, KeySort) :=

    se ? sdu: SeSduSort [(xPri(sdu) eq Inp) and
                                   (xFn(sdu) eq SE_Accept)];
    exit (reqset, resset, outstaskwaits,
           regset, outstinquires, outstattends,
           Insert(TPair(xCallerKey(sdu), xType(sdu)), outstaccepts),
                  generator)
           (* ...Note the outstanding SE_Accept *)

endproc (* I_ACCEPT *)


(* ***********************************************************
 * Component class: functional component/stopwatch component
 * Comments: This component specifies both the functional and
 * stopwatch aspects of the SE_Answer Input event. An SE_Answer
 * Input conveys a response pkt targeted at an SE_Ask_Wait.
 * Therefore the occurrence time of the SE_Answer Input
 * corresponds to the arrival time of the response pkt. We note
 * this time and store it, together with the response pkt, in
 * the resset data structure. (For the purposes of this
 * specification, the response 'arrival time' is taken to equal
 * response pkt 'available time' -- the time at which an
 * SE_Ask_Wait can take the response pkt.) The response pkt
 * arrival time is used to calculate if the SE_Ask_Wait has timed
 * out before the response arrived.
 *********************************************************** *)

process I_ANSWER [se] (reqset: ReqsSetSort,
                               resset: ResSetSort,
                               outstaskwaits: OutStAskWaitsSort,
                               regset: RegDBaseSort,
                               outstinquires: OutStInquiresSort,
                               outstattends: OutStAttendsSort,
                               outstaccepts: OutStAcceptsSort,
                          generator: KeySort)
: exit (ReqsSetSort, ResSetSort, OutStAskWaitsSort, RegDBaseSort,
    OutStInquiresSort, OutStAttendsSort,
    OutStAcceptsSort, KeySort) :=

    se ? sdu: SeSduSort [(xPri(sdu) eq Inp) and
                          (xFn(sdu) eq SE_Answer)]
                          @t1;
              (* ...Note the arrival time of the SE_Answer Input,
                     because this arrival time is needed to
                     constrain the Output event of the
                     SE_Ask_Wait invocation at which
                     this SE_Answer is aimed *)

    exit (reqset,
          Insert(ResPkt(xCallerKey(sdu), xPdu(sdu), t1), resset),
          outstaskwaits, regset, outstinquires, outstattends,
          outstaccepts, generator)
          (* ...Create a response pkt *)

endproc (* I_ANSWER *)


(* ************************************************************
 * Component class: functional component
 * Comments: Constrains and processes the Input event occurrance
 * for SE_Tell.
 ************************************************************ *)

process I_TELL [se] (reqset: ReqsSetSort,
                     resset: ResSetSort,
                     outstaskwaits: OutStAskWaitsSort,
                     regset: RegDBaseSort,
                     outstinquires: OutStInquiresSort,
                     outstattends: OutStAttendsSort,
                     outstaccepts: OutStAcceptsSort,
                     generator: KeySort)
: exit (ReqsSetSort, ResSetSort, OutStAskWaitsSort, RegDBaseSort,
    OutStInquiresSort, OutStAttendsSort,
    OutStAcceptsSort, KeySort) :=

    se ? sdu: SeSduSort [(xPri(sdu) eq Inp) and
                          (xFn(sdu) eq SE_Tell)];
    exit (Insert(ReqPkt(xCallerKey(sdu), xCallerkey(sdu),
                 xPdu(sdu), xType(sdu)), reqset),
          resset, outstaskwaits,
          regset, outstinquires, outstattends,
          outstaccepts, generator)
          (* ...Create a request packet *)

endproc (* I_TELL *)


(* ************************************************************
 * Component class: storage component
 * Comments: Note, in the registration database (regset), that
 * an SE_Service_User is going off-line.
 ************************************************************ *)

process I_TERMINATE [se] (reqset: ReqsSetSort,
                          resset: ResSetSort,
                          outstaskwaits: OutStAskWaitsSort,
                          regset: RegDBaseSort,
                          outstinquires: OutStInquiresSort,
                          outstattends: OutStAttendsSort,
                          outstaccepts: OutStAcceptsSort,
                          generator: KeySort)
```

```
      : exit (ReqsSetSort, ResSetSort, OutStAskWaitsSort, RegDBaseSort,
          OutStInquiresSort, OutStAttendsSort,
          OutStAcceptsSort, KeySort) :=

        se ? sdu: SeSduSort [(xPri(sdu) eq Inp) and
                                  (xFn(sdu) eq SE_Terminate)];
        ([xTemp(sdu) eq Cold] -> (* Cold terminate *)
            exit (reqset,
                  RemoveByKey(xCallerKey(sdu), resset),
                  outstaskwaits,
                  RemoveByKey(xCallerKey(sdu), regset),
                  outstinquires, outstattends,
                  outstaccepts, generator)
                  (* ...Remove all response messages to this entity*)
                  (* ...Unregister this entity *)
                  (* QUESTION TO FRB: Does it matter? If it does
                     then what about response messages targeted at
                     this DS_Service_Key in the future ? *)
         []
          [xTemp(sdu) eq Warm] -> (* Warm terminate *)
             exit (reqset, resset, outstaskwaits, regset,
                  outstinquires, outstattends,
                  outstaccepts, generator)
                  (* Preserve the registration entry *)
         )

      endproc (* I_TERMINATE *)


endproc (* INPUT *)

   (* -------------------------------------------------------------
    * Component class: functional component
    * Comments: OUTPUT deals with primarily functional constraints
    * over events which represent Output SE SDUs.
    ------------------------------------------------------------- *)

   process OUTPUT [se] (reqset: ReqsSetSort,
                resset: ResSetSort,
                outstaskwaits: OutStAskWaitsSort,
                regset: RegDBaseSort,
                outstinquires: OutStInquiresSort,
                outstattends: OutStAttendsSort,
                outstaccepts: OutStAcceptsSort,
                generator: KeySort)
   : exit (ReqsSetSort, ResSetSort, OutStAskWaitsSort, RegDBaseSort,
       OutStInquiresSort, OutStAttendsSort, OutStAcceptsSort,
       KeySort) :=

      (* Separate LOTOS process templates to impose constrains on and
       * deal with the processing for each type of
       * SE_Callable_Function Output event...
       *)

      (
         O_INITIALIZE[se](reqset, resset, outstaskwaits, regset,
                          outstinquires, outstattends, outstaccepts,
                          generator)
      [] O_INQUIRE_KEY[se](reqset, resset, outstaskwaits, regset,
                          outstinquires, outstattends, outstaccepts,
                          generator)
      [] O_ASK_WAIT[se](reqset, resset, outstaskwaits, regset,
                       outstinquires, outstattends, outstaccepts,
                       generator,TIMEOUT_PERIOD_VALUE)
                       (* the placeholder TIMEOUT_PERIOD_VALUE
                        * should be actualized to an appropriate
                        * TimeSort value *)
    [] O_ATTEND[se](reqset, resset, outstaskwaits, regset,
                        outstinquires, outstattends, outstaccepts,
                        generator)
    [] O_ACCEPT[se](reqset, resset, outstaskwaits, regset,
                        outstinquires, outstattends, outstaccepts,
                        generator)
    [] O_ANSWER[se](reqset, resset, outstaskwaits, regset,
                        outstinquires, outstattends,
                        outstaccepts, generator)
    [] O_TELL[se](reqset, resset, outstaskwaits, regset,
                    outstinquires, outstattends,
                    outstaccepts, generator)
    [] O_TERMINATE[se](reqset, resset, outstaskwaits, regset,
                        outstinquires, outstattends,
                        outstaccepts, generator)
    )

where

    (* ************************************************************
     * Component class: functional component
     * Comments: Constrains and processes the Output event occurrence
     * for SE_Initialize.
     ************************************************************ *)

    process O_INITIALIZE [se] (reqset: ReqsSetSort,
                                resset: ResSetSort,
                                outstaskwaits: OutStAskWaitsSort,
                                regset: RegDBaseSort,
                                outstinquires: OutStInquiresSort,
                                outstattends: OutStAttendsSort,
                                outstaccepts: OutStAcceptsSort,
                                generator: KeySort)
    : exit (ReqsSetSort, ResSetSort, OutStAskWaitsSort, RegDBaseSort,
         OutStInquiresSort, OutStAttendsSort, outStAcceptsSort,
         KeySort) :=

         se ? sdu: SeSduSort [(xPri(sdu) eq Out) and
                                   (xFn(sdu) eq SE_Initialize) and
                              (NPair(xNKey(sdu), xName(sdu))
                                   IsIn regset) and
                              (xRtnCode(sdu) eq SE_Ok)
                              ];
                              (* ...The Name_Key_Pair must be an
                                   entry in the Registration-Database *)
       exit (reqset, resset, outstaskwaits, regset,
             outstinquires, outstattends, outstaccepts, generator)

    endproc (* O_INITIALIZE *)

    (* ************************************************************
     * Component class: functional component
     * Comments: Constrains and processes the Output event occurrence
     * for SE_Inquire_Key.
     ************************************************************ *)

    process O_INQUIRE_KEY [se] (reqset: ReqsSetSort,
                                resset: ResSetSort,
                                outstaskwaits: OutStAskWaitsSort,
                                regset: RegDBaseSort,
                                outstinquires: OutStInquiresSort,
                                outstattends: OutStAttendsSort,
                                outstaccepts: OutStAcceptsSort,
                                generator: KeySort)
    : exit (ReqsSetSort, ResSetSort, OutStAskWaitsSort, RegDBaseSort,
    OutStInquiresSort, OutStAttendsSort,
    OutStAcceptsSort, KeySort) :=

   se ? sdu: SeSduSort [(xPri(sdu) eq Out) and
                               (xFn(sdu) eq SE_Inquire_Key) and
                               (xQKey(sdu) eq
                                FindKey(
                                   RtnInqNameByKey(xCallerKey(sdu),
                                                          outstinquires),
                                   regset)) and
                               (xRtnCode(sdu) eq SE_Ok)
                              ];
          (* ...The queried Name form the SE_Inquire request is
           * used to search the Registration-Database for its
           * complementary IIS_Service_Key
           *)

   exit (reqset, resset, outstaskwaits, regset,
       RemoveByKey(xCallerKey(sdu), outstinquires),
       outstattends, outstaccepts, generator)
       (* ...Remove the outstanding request *)

endproc (* O_INQUIRE_KEY *)


(* **********************************************************
 * Component class: functional component/timeout component
 * Comments: The functional and timeout constraints for
 * SE_Ask_Wait Outputs, are quite integrated --- this component
 * is responsible for imposing both.
 *
 * For the concerns of this specification and this component,
 * there are 3 different scenarios in which SE_Ask_Wait Output
 * events may occur:
 *
 * 1) an appropriate response pkt to the SE_Ask_Wait Input
 * arrives before the timeout expires =>
 * Output event (with RtnCode = SE_Ok) offered from arrival time
 * of resonse pkt.
 *
 * 2) timeout expires before appropriate response pkt arrives,
 * but then appropriate response pkt arrives before the
 * SE_Ask_Wait Output event occurs =>
 * Output event (with RtnCode = SE_Timeout) offered from end
 * of timeout period.
 *
 * 3) timeout expires before appropriate response pkt arrives,
 * and no approriate response pkt arrives before the
 * SE_Ask_Wait Output event occurs =>
 * Output event (with RtnCode = SE_Timeout) offered from end
 * of timeout period.
 *
 * Note: a scenario-3 may turn into a scenario-2.
 ********************************************************** *)

process O_ASK_WAIT [se] (reqset: ReqsSetSort,
                              resset: ResSetSort,
                              outstaskwaits: OutStAskWaitsSort,
                              regset: RegDBaseSort,
                              outstinquires: OutStInquiresSort,
                              outstattends: OutStAttendsSort,
                              outstaccepts: OutStAcceptsSort,
                              generator: KeySort,
                              timeout_period:TimeSort)
: exit (ReqsSetSort, ResSetSort, OutStAskWaitsSort, RegDBaseSort,
    OutStInquiresSort, OutStAttendsSort,
    OutStAcceptsSort, KeySort) :=
   OutStInquiresSort, OutStAttendsSort,
   OutStAcceptsSort, KeySort) :=

  se ? sdu: SeSduSort [(xPri(sdu) eq Out) and
                       (xFn(sdu) eq SE_Inquire_Key) and
                       (xQKey(sdu) eq
                        FindKey(
                          RtnInqNameByKey(xCallerKey(sdu),
                                          outstinquires),
                          regset)) and
                       (xRtnCode(sdu) eq SE_Ok)
                      ];
        (* ...The queried Name form the SE_Inquire request is
         * used to search the Registration-Database for its
         * complementary IIS_Service_Key
         *)

  exit (reqset, resset, outstaskwaits, regset,
      RemoveByKey(xCallerKey(sdu), outstinquires),
      outstattends, outstaccepts, generator)
      (* ...Remove the outstanding request *)

endproc (* O_INQUIRE_KEY *)


(* ********************************************************
 * Component class: functional component/timeout component
 * Comments: The functional and timeout constraints for
 * SE_Ask_Wait Outputs, are quite integrated --- this component
 * is responsible for imposing both.
 *
 * For the concerns of this specification and this component,
 * there are 3 different scenarios in which SE_Ask_Wait Output
 * events may occur:
 *
 * 1) an appropriate response pkt to the SE_Ask_Wait Input
 * arrives before the timeout expires =>
 * Output event (with RtnCode = SE_Ok) offered from arrival time
 * of resonse pkt.
 *
 * 2) timeout expires before appropriate response pkt arrives,
 * but then appropriate response pkt arrives before the
 * SE_Ask_Wait Output event occurs =>
 * Output event (with RtnCode = SE_Timeout) offered from end
 * of timeout period.
 *
 * 3) timeout expires before appropriate response pkt arrives,
 * and no approriate response pkt arrives before the
 * SE_Ask_Wait Output event occurs =>
 * Output event (with RtnCode = SE_Timeout) offered from end
 * of timeout period.
 *
 * Note: a scenario-3 may turn into a scenario-2.
 ******************************************************** *)

process O_ASK_WAIT [se] (reqset: ReqsSetSort,
                         resset: ResSetSort,
                         outstaskwaits: OutStAskWaitsSort,
                         regset: RegDBaseSort,
                         outstinquires: OutStInquiresSort,
                         outstattends: OutStAttendsSort,
                         outstaccepts: OutStAcceptsSort,
                         generator: KeySort,
                         timeout_period:TimeSort)
: exit (ReqsSetSort, ResSetSort, OutStAskWaitsSort, RegDBaseSort,
     OutStInquiresSort, OutStAttendsSort,
     OutStAcceptsSort, KeySort) :=
(
    (* If an appropriate reponse packet arrives within the
     * timeout period...
     *)
    se ? sdu: SeSduSort
         [(xPri(sdu) eq Out) and
          (xFn(sdu) eq SE_Ask_Wait) and
          (AskWaitTriple(xCallerKey(sdu), invoke_time,
                                                  timeout_period)
               IsIn outstaskwaits) and
          (ResPkt(xCallerKey(sdu), xPdu(sdu), res_arr_time)
               IsIn resset) and
          (res_arr_time le (invoke_time + timeout_period)) and
          (xRtnCode(sdu) eq SE_Ok)
         ]
         {setGE(res_arr_time)};
              (* ...Check:
               * 1) existing outstanding SE_Ask_Wait, and
               * 2) existing appropriate response PDU, and
               * 3) the response pkt has arrived before the
               * timeout_period expires
               *)
    exit (reqset,
          Remove(ResPkt(xCallerKey(sdu), xPdu(sdu),
                 res_arr_time), resset),
          Remove(AskWaitTriple(xCallerKey(sdu),invoke_time,
                                                  timeout_period),
                 outstaskwaits),
          regset, outstinquires, outstattends, outstaccepts,
          generator)
              (* ...Remove the response PDU *)
              (* ...Remove the outstanding SE_Ask_Wait *)
[]
    (* If the SE_Ask_Wait times-out but an appropriate reponse
     * packet then arrives after the timeout period...
     *)
    se ? sdu: SeSduSort
         [(xPri(sdu) eq Out) and
          (xFn(sdu) eq SE_Ask_Wait) and
          (AskWaitTriple(xCallerKey(sdu), invoke_time,
                                                  timeout_period)
               IsIn outstaskwaits) and
          (ResPkt(xCallerKey(sdu), xPdu(sdu), res_arr_time)
               IsIn resset) and
          (res_arr_time gt (invoke_time + timeout_period)) and
          (xRtnCode(sdu) eq SE_Timeout)
         ]
         {setGT(invoke_time + timeout_period)};
              (* ...Check:
               * 1) existing outstanding SE_Ask_Wait, and
               * 2) existing appropriate response PDU, and
               * 3) the response pkt has arrived after the
               * timeout_period expires
               *)
    exit (reqset,
          Remove(ResPkt(xCallerKey(sdu), xPdu(sdu),
                 res_arr_time), resset),
          Remove(AskWaitTriple(xCallerKey(sdu),invoke_time,
                                                  timeout_period),
                 outstaskwaits),
          regset, outstinquires, outstattends, outstaccepts,
          generator)
              (* ...Remove the response PDU *)
              (* ...Remove the outstanding SE_Ask_Wait *)
              (* QUESTION TO FRB: should we really remove the
                 response pkt in this case? *)
```

```
   []
     (* If the SE_Ask_Wait times-out (and no appropriate reponse
      * packet has arrived (yet) after the timeout period...
      *)
     se ? sdu: SeSduSort
           [(xPri(sdu) eq Out) and
            (xFn(sdu) eq SE_Ask_Wait) and
            (AskWaitTriple(xCallerKey(sdu), invoke_time,
                                                   timeout_period)
                 IsIn outstaskwaits) and
            (Not(ResPkt(xCallerKey(sdu), pdu, res_arr_time)
                       IsIn resset)) and
            (xRtnCode(sdu) eq SE_Timeout) and
            (xPdu(sdu) eq ErrorPdu)
           ]
           {setGT(invoke_time + timeout_period)};
                (* ...Check:
                 * 1) existing outstanding SE_Ask_Wait, and
                 * 2) no appropriate existing response PDU
                 *)
     exit (reqset, resset,
            Remove(AskWaitTriple(xCallerKey(sdu),invoke_time,
                                                       timeout_period),
                   outstaskwaits),
            regset, outstinquires, outstattends, outstaccepts,
            generator)
                (* ...Remove the outstanding SE_Ask_Wait *)
   )

 endproc (* O_ASK_WAIT *)


(* *************************************************************
 * Component class: functional component
 * Comments: Constrains and processes the Output event occurrance
 * for SE_Attend.
 ************************************************************* *)

process O_ATTEND [se] (reqset: ReqsSetSort,
                        resset: ResSetSort,
                        outstaskwaits: OutStAskWaitsSort,
                        regset: RegDBaseSort,
                        outstinquires: OutStInquiresSort,
                        outstattends: OutStAttendsSort,
                        outstaccepts: OutStAcceptsSort,
                        generator: KeySort)
: exit (ReqsSetSort, ResSetSort, OutStAskWaitsSort, RegDBaseSort,
     OutStInquiresSort, OutStAttendsSort,
     OutStAcceptsSort, KeySort) :=

   se ? sdu: SeSduSort [(xPri(sdu) eq Out) and
                         (xFn(sdu) eq SE_Attend) and
                         (xType(sdu) eq RtnAType(
                               xTPairByKey(xCallerKey(sdu),
                                           outstattends),
                               reqset)) and
                         (xRtnCode(sdu) eq SE_Ok)
                        ];
              (* ...Check if a request packet for the
                    given Key_Type pair exists -- return
                    that Type if it does, else return 0 *)
   exit (reqset, resset, outstaskwaits,
          regset, outstinquires,
          Remove(TPair(xCallerKey(sdu), xType(sdu)), outstattends),
          outstaccepts, generator)
              (* ...Remove the outstanding SE_Attend *)
```

```
endproc (* O_ATTEND *)

(* ************************************************************
 * Component class: functional component
 * Comments: Constrains and processes the Output event occurrance
 * for SE_Accept.
 ************************************************************ *)

process O_ACCEPT [se] (reqset: ReqsSetSort,
                       resset: ResSetSort,
                       outstaskwaits: OutStAskWaitsSort,
                       regset: RegDBaseSort,
                       outstinquires: OutStInquiresSort,
                       outstattends: OutStAttendsSort,
                       outstaccepts: OutStAcceptsSort,
                       generator: KeySort)
: exit (ReqsSetSort, ResSetSort, OutStAskWaitsSort, RegDBaseSort,
    OutStInquiresSort, OutStAttendsSort,
    OutStAcceptsSort, KeySort) :=

  choice reqpkt: ReqPktSort [] (* ..Choose any request packet *)
    [(reqpkt IsIn reqset) and
     (TPair(xTargetKey(reqpkt), xType(reqpkt)) IsIn outstaccepts)
    ] -> (* .. Constrain the choice to
          * be a packet in the reqset
          * and for this packet to be
          * targeted at an entity which
          * has an Outstanding_SE_Accept
          *)
        (se ? sdu: SeSduSort [(xPri(sdu) eq Out) and
                               (xFn(sdu) eq SE_Accept) and
                               (xInitiatorKey(sdu) eq
                                   xInitiatorKey(reqpkt)) and
                               (xPdu(sdu) eq xPdu(reqpkt)) and
                               (xRtnCode(sdu) eq SE_Ok)
                              ];
                  (* ...And return this request packet *)

         exit (Remove(reqpkt, reqset),
               resset, outstaskwaits,
               regset, outstinquires,
               outstattends,
               RemoveByKey(xCallerKey(sdu), outstaccepts),
               generator)
                     (* .. Remove the request packet *)
                  (* .. Remove the outstanding SE_Accept *)
        )

endproc (* O_ACCEPT *)

(* ************************************************************
 * Component class: functional component
 * Comments: Constrains and processes the Output event occurrance
 * for SE_Answer.
 ************************************************************ *)

process O_ANSWER [se] (reqset: ReqsSetSort,
                       resset: ResSetSort,
                       outstaskwaits: OutStAskWaitsSort,
                       regset: RegDBaseSort,
                       outstinquires: OutStInquiresSort,
                       outstattends: OutStAttendsSort,
                       outstaccepts: OutStAcceptsSort,
                       generator: KeySort)
: exit (ReqsSetSort, ResSetSort, OutStAskWaitsSort, RegDBaseSort,
      OutStInquiresSort, OutStAttendsSort, outStAcceptsSort,
      KeySort) :=

      se ? sdu: SeSduSort [(xPri(sdu) eq Out) and
                                  (xFn(sdu) eq SE_Answer) and
                               (xRtnCode(sdu) eq SE_Ok)];
     exit (reqset, resset, outstaskwaits, regset,
            outstinquires, outstattends, outstaccepts, generator)

  endproc (* O_ANSWER *)

  (* ***********************************************************
   * Component class: functional component
   * Comments: Constrains and processes the Output event occurrance
   * for SE_Tell.
   ************************************************************ *)

  process O_TELL [se] (reqset: ReqsSetSort,
                               resset: ResSetSort,
                               outstaskwaits: OutStAskWaitsSort,
                               regset: RegDBaseSort,
                               outstinquires: OutStInquiresSort,
                               outstattends: OutStAttendsSort,
                               outstaccepts: OutStAcceptsSort,
                               generator: KeySort)
  : exit (ReqsSetSort, ResSetSort, OutStAskWaitsSort, RegDBaseSort,
      OutStInquiresSort, OutStAttendsSort, outStAcceptsSort,
      KeySort) :=

      se ? sdu: SeSduSort [(xPri(sdu) eq Out) and
                                  (xFn(sdu) eq SE_Tell) and
                               (xRtnCode(sdu) eq SE_Ok)];
     exit (reqset, resset, outstaskwaits, regset,
            outstinquires, outstattends, outstaccepts, generator)

  endproc (* O_TELL *)

  (* ***********************************************************
   * Component class: functional component
   * Comments: Constrains and processes the Output event occurrance
   * for SE_Terminate.
   ************************************************************ *)

  process O_TERMINATE [se] (reqset: ReqsSetSort,
                               resset: ResSetSort,
                               outstaskwaits: OutStAskWaitsSort,
                               regset: RegDBaseSort,
                               outstinquires: OutStInquiresSort,
                               outstattends: OutStAttendsSort,
                               outstaccepts: OutStAcceptsSort,
                               generator: KeySort)
  : exit (ReqsSetSort, ResSetSort, OutStAskWaitsSort, RegDBaseSort,
      OutStInquiresSort, OutStAttendsSort, outStAcceptsSort,
      KeySort) :=

      se ? sdu: SeSduSort [(xPri(sdu) eq Out) and
                                  (xFn(sdu) eq SE_Terminate) and
                               (xRtnCode(sdu) eq SE_Ok)];
     exit (reqset, resset, outstaskwaits, regset,
            outstinquires, outstattends, outstaccepts, generator)

  endproc (* O_TERMINATE *)

endproc (* OUTPUT *)
```

```
        endproc (* IO *)
    endproc (* FUNCTIONALITY *)
endspec (* SE_SERVICE *)
```

Appendix C

TLOTOS pre-defined library data types

This appendix contains ACT ONE data types for inclusion in the pre-defined data types library of TLOTOS. The data types define all the functions and sorts pertaining to TLOTOS time. This is reference material for section 6.5.1.

```
specification TestTimeLibrary : noexit

(* ************************************************************************
 * Build upon the standard library.
 ************************************************************************ *)
library
   NaturalNumber
endlib

(* ************************************************************************
 * TimeType: defines time values which are isomorphic to the natural numbers.
 ************************************************************************ *)
type TimeType is NaturalNumber renamedby
   sortnames
      TimeSort for Nat
endtype (* TimeType *)

(* ************************************************************************
 * For convenience while testing.
 ************************************************************************ *)
type TestingTimeType is TimeType
   opns
      v0, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10: -> TimeSort
   eqns
      ofsort TimeSort
      v0 = 0;
      v1 = succ(v0);
      v2 = succ(v1);
      v3 = succ(v2);
      v4 = succ(v3);
      v5 = succ(v4);
      v6 = succ(v5);
      v7 = succ(v6);
```

```
        v8 = succ(v7);
        v9 = succ(v8);
        v10 = succ(v9);
endtype (* TestingTimeType *)


(* ***************************************************************************
 * TimeSetType: defines non-denumerable sets of time values. Also, specifies the
 * functions which, for the sake of must-timing, ASAP and ALAP time-policies,
 * require to be definable for all terms of TimeSetSort.
 ************************************************************************** *)
type TimeSetType is TimeType

    sorts
        TimeSetSort

    opns
        (* The first primitive constructor function... *)
        Empty: -> TimeSetSort
        (*
         * Functions which must be definable for all terms of TimeSetSort...
         *)
        _ IsIn _, _ isGTAllMembersOf _ : TimeSort, TimeSetSort -> Bool
        isUpperLimited : TimeSetSort -> Bool
        Min, Max : TimeSetSort -> TimeSort

    eqns
      forall t, t1: TimeSort, s1, s2, s3:TimeSetSort

          ofsort Bool
              (*
               * Define IsIn, isGTAllMembersOf and isUpperLimited for Empty
               *)
              t IsIn Empty = False;
              t isGTAllMembersOf Empty = True;
              isUpperLimited(Empty) = True;

          ofsort TimeSort
              (*
               * Define Min and Max for Empty.
               * (For completeness only, these instantiations are not
               * used in TLOTOS semantics.)
               *)
              Min(Empty) = 0;
              Max(Empty) = 0;

endtype (* TimeSetType *)


(* ***************************************************************************
 * SetGeneratorType: defines all the primitive constructors (except Empty)
 * of TimeSetSort terms. The complete list of TimeSetSort primitive
 * constructors is: Empty, setEQ, setLE, setGE, setInterval and Union.
 *
 * The functions IsIn, isGTAllMembersOf, isUpperLimited,
 * Min and Max are defined for all TimeSetSort terms constructed using the
 * primitive constructor functions.
 *
 * The constructor functions setLT, setGT and Intersection are defined in
 * terms of the primitive constructors. (Some rationalizations of primitive
 * constructor terms has been defined. Actually, it would be possible
 * to completely rationalize out certain 'primitive' constructor functions
 * altogether.)
 *
 * The idea is that TLOTOS users can define new TimeSetSort generator
* functions in terms of the primitive functions. Then these new TimeSetSort
* generator will be reducable to combinations of primitive constructors.
* Hence, in effect, the IsIn, etc. functions will implicitly be defined
* over the new TimeSetSort generator functions.
*
************************************************************************ *)
type SetGeneratorFunctionsType is TimeSetType

   opns
      (*
       * More primitive constructor functions...
       *
       * (We define setGT, setLT and Intersection in
       * terms of setEQ, setLE, setGE, setInterval and Union)
       *)
      setLE, setGE,
      setEQ, setLT, setGT: TimeSort -> TimeSetSort
      setInterval: TimeSort, TimeSort -> TimeSetSort
      _ Intersection _,
      _ Union _ : TimeSetSort, TimeSetSort -> TimeSetSort

   eqns
      forall t1,t2,t3,t4: TimeSort, s1,s2,s3,s4: TimeSetSort

         ofsort TimeSetSort

            (*
             * Trivally rationalise setInterval
             *
             *)
            t2 lt t1 => setInterval(t1,t2) = Empty;
            t2 eq t1 => setInterval(t1,t2) = setEQ(t1);

            (*
             * Rationalize Union
             * involving Empty
             *)
            s1 Union Empty = s1;
            Empty Union s1 = s1;

            (*
             * Rewrite setLT and setGT
             * in terms of setLE and setGE
             *)
            setLT(succ(t1)) = setLE(t1);
            setLT(0) = Empty;
            setGT(t1) = setGE(succ(t1));

            (*
             * Remove Union from terms
             * involving setEQ
             *)
            t1 eq t2 => setEQ(t1) Union setEQ(t2) = setEQ(t1);

            t1 le t2 => setEQ(t1) Union setLE(t2) = setLE(t2);
            t1 le t2 => setLE(t2) Union setEQ(t1) = setLE(t2);

            t1 ge t2 => setEQ(t1) Union setGE(t2) = setGE(t2);
            t1 ge t2 => setGE(t2) Union setEQ(t1) = setGE(t2);

            (t1 ge t2) and (t1 le t3) =>
                  setEQ(t1) Union setInterval(t2,t3) = setInterval(t2,t3);
            (t1 ge t2) and (t1 le t3) =>
                  setInterval(t2,t3) Union setEQ(t1) = setInterval(t2,t3);

            (*
 * Remove Union from terms
 * involving setLE, setGE
 *)
t1 lt t2 ⇒ setLE(t1) Union setLE(t2) = setLE(t2);
not(t1 lt t2) ⇒ setLE(t1) Union setLE(t2) = setLE(t1);
t1 gt t2 ⇒ setGE(t1) Union setGE(t2) = setGE(t2);
not(t1 gt t2) ⇒ setGE(t1) Union setGE(t2) = setGE(t1);

t1 ge t3 ⇒
    setLE(t1) Union setInterval(t2,t3) = setLE(t1);
t1 ge t3 ⇒
    setInterval(t2,t3) Union setLE(t1) = setLE(t1);
(t1 ge t2) and (t1 lt t3) ⇒
    setLE(t1) Union setInterval(t2,t3) = setLE(t3);
(t1 ge t2) and (t1 lt t3) ⇒
    setInterval(t2,t3) Union setLE(t1) = setLE(t3);

t1 le t2 ⇒
    setGE(t1) Union setInterval(t2,t3) = setGE(t1);
t1 le t2 ⇒
    setInterval(t2,t3) Union setGE(t1) = setGE(t1);
(t1 le t3) and (t1 gt t2) ⇒
    setGE(t1) Union setInterval(t2,t3) = setGE(t2);
(t1 le t3) and (t1 gt t2) ⇒
    setInterval(t2,t3) Union setGE(t1) = setGE(t2);

(*
 * Remove Union from terms
 * involving setInterval
 *)
(t1 le t3) and (t2 ge t4) ⇒
    setInterval(t1,t2) Union setInterval(t3,t4) = setInterval(t1,t2);
(t3 lt t1) and (t4 gt t2) ⇒
    setInterval(t1,t2) Union setInterval(t3,t4) = setInterval(t3,t4);
(t1 le t3) and (t2 lt t4) and (t2 ge t3) ⇒
    setInterval(t1,t2) Union setInterval(t3,t4) = setInterval(t1,t4);
(t1 gt t3) and (t2 ge t4) and (t4 ge t1) ⇒
    setInterval(t1,t2) Union setInterval(t3,t4) = setInterval(t3,t2);


(*
 * Rationalize Intersection
 * involving Empty
 *)
Empty Intersection s1 = Empty;
s1 Intersection Empty = Empty;

(*
 * Remove Intersection from terms
 * involving setEQ
 *)
t1 eq t2 ⇒ setEQ(t1) Intersection setEQ(t2) = setEQ(t1);
not(t1 eq t2) ⇒ setEQ(t1) Intersection setEQ(t2) = Empty;

t1 le t2 ⇒ setEQ(t1) Intersection setLE(t2) = setEQ(t1);
not(t1 le t2) ⇒ setEQ(t1) Intersection setLE(t2) = Empty;
t1 le t2 ⇒ setLE(t2) Intersection setEQ(t1) = setEQ(t1);
not(t1 le t2) ⇒ setLE(t2) Intersection setEQ(t1) = Empty;

t1 ge t2 ⇒ setEQ(t1) Intersection setGE(t2) = setEQ(t1);
not(t1 ge t2) ⇒ setEQ(t1) Intersection setGE(t2) = Empty;
t1 ge t2 ⇒ setGE(t2) Intersection setEQ(t1) = setEQ(t1);
not(t1 ge t2) ⇒ setGE(t2) Intersection setEQ(t1) = Empty;

(t1 ge t2) and (t1 le t3) ⇒
      setEQ(t1) Intersection setInterval(t2,t3) = setEQ(t1);
```

```
not((t1 ge t2) and (t1 le t3)) ⇒
      setEQ(t1) Intersection setInterval(t2,t3) = Empty;
(t1 ge t2) and (t1 le t3) ⇒
      setInterval(t2,t3) Intersection setEQ(t1) = setEQ(t1);
not((t1 ge t2) and (t1 le t3)) ⇒
      setInterval(t2,t3) Intersection setEQ(t1) = Empty;

(*
 * Remove Intersection from terms
 * involving setLE, setGE
 *)
t1 lt t2 ⇒ setLE(t1) Intersection setLE(t2) = setLE(t1);
not(t1 lt t2) ⇒ setLE(t1) Intersection setLE(t2) = setLE(t2);
t1 gt t2 ⇒ setGE(t1) Intersection setGE(t2) = setGE(t1);
not(t1 gt t2) ⇒ setGE(t1) Intersection setGE(t2) = setLE(t2);

t1 lt t2 ⇒
     setLE(t1) Intersection setGE(t2) = Empty;
not(t1 lt t2) ⇒
     setLE(t1) Intersection setGE(t2) = setInterval(t2,t1);
t1 lt t2 ⇒
     setGE(t2) Intersection setLE(t1) = Empty;
not(t1 lt t2) ⇒
     setGE(t2) Intersection setLE(t1) = setInterval(t2,t1);

t1 ge t3 ⇒
     setLE(t1) Intersection setInterval(t2,t3) = setInterval(t2,t3);
t1 ge t3 ⇒
     setInterval(t2,t3) Intersection setLE(t1) = setInterval(t2,t3);
(t1 ge t2) and (t1 lt t3) ⇒
     setLE(t1) Intersection setInterval(t2,t3) = setInterval(t2,t1);
(t1 ge t2) and (t1 lt t3) ⇒
     setInterval(t2,t3) Intersection setLE(t1) = setInterval(t2,t1);
(t1 lt t2) ⇒
     setLE(t1) Intersection setInterval(t2,t3) = Empty;
(t1 lt t2) ⇒
     setInterval(t2,t3) Intersection setLE(t1) = Empty;

t1 le t2 ⇒
     setGE(t1) Intersection setInterval(t2,t3) = setInterval(t2,t3);
t1 le t2 ⇒
     setInterval(t2,t3) Intersection setGE(t1) = setInterval(t2,t3);
(t1 le t3) and (t1 gt t2) ⇒
     setGE(t1) Intersection setInterval(t2,t3) = setInterval(t1,t3);
(t1 le t3) and (t1 gt t2) ⇒
     setInterval(t2,t3) Intersection setGE(t1) = setInterval(t1,t3);
(t1 gt t3) ⇒
     setGE(t1) Intersection setInterval(t2,t3) = Empty;
(t1 gt t3) ⇒
     setInterval(t2,t3) Intersection setGE(t1) = Empty;

(*
 * Remove Intersection from terms
 * involving setInterval
 *)
(t1 le t3) and (t2 ge t4) ⇒
     setInterval(t1,t2) Intersection setInterval(t3,t4)
          = setInterval(t3,t4);
(t3 lt t1) and (t4 gt t2) ⇒
     setInterval(t1,t2) Intersection setInterval(t3,t4)
          = setInterval(t1,t2);
(t1 le t3) and (t2 lt t4) and (t2 ge t3) ⇒
     setInterval(t1,t2) Intersection setInterval(t3,t4)
          = setInterval(t3,t2);
(t1 gt t3) and (t2 ge t4) and (t4 ge t1) ⇒
     setInterval(t1,t2) Intersection setInterval(t3,t4)
```

```
            = setInterval(t1,t4);
    (t2 lt t3) or (t1 gt t4) =>
        setInterval(t1,t2) Intersection setInterval(t3,t4)
            = Empty;


    (*
     * Remove Intersection from Union combinations
     *)
    (s1 Union s2) Intersection s3 =
        (s1 Intersection s3) Union (s2 Intersection s3);
    s1 Intersection (s3 Union s4) =
        (s1 Intersection s3) Union (s1 Intersection s4);


(*
 * Now define isUpperLimited, IsIn, isGTAllMembersOf
 * for terms constructed from setEQ, setLE, setGE,
 * setInterval and Union
 *)

ofsort Bool

    (*
     * Define isUpperlimited
     * for setEQ, setLE, setGE, setInterval, Union
     *)
    isUpperLimited(setEQ(t1)) = True;
    isUpperLimited(setLE(t1)) = True;
    isUpperLimited(setGE(t1)) = False;
    isUpperLimited(setInterval(t1,t2)) = True;
    isUpperLimited(s1 union s2)
        = isUpperLimited(s1) and isUpperLimited(s2);

    (*
     * Define IsIn
     * for setEQ, setLE, setGE, setInterval, Union
     *)
    t1 eq t2 => t1 IsIn setEQ(t2) = True;
    not(t1 eq t2) => t1 IsIn setEQ(t2) = False;

    t1 le t2 => t1 IsIn setLE(t2) = True;
    not(t1 lt t2) => t1 IsIn setLE(t2) = False;

    t1 ge t2 => t1 IsIn setGE(t2) = True;
    not(t1 ge t2) => t1 IsIn setGE(t2) = False;

    (t1 ge t2) and (t1 le t3) =>
        t1 IsIn setInterval(t2,t3) = True;
    not((t1 ge t2) and (t1 le t3)) =>
        t1 IsIn setInterval(t2,t3) = False;

    t1 IsIn (s1 union s2) = (t1 IsIn s1) or (t1 IsIn s2);


    (*
     * Define isGTAllMembersOf
     * for setEQ, setLE, setGE, setInterval, Union
     *)
    t1 eq t2 => t1 isGTAllMembersOf setEQ(t2) = True;
    not(t1 eq t2) => t1 isGTAllMembersOf setEQ(t2) = False;

    t1 gt t2 => t1 isGTAllMembersOf setLE(t2) = True;
    not(t1 gt t2) => t1 isGTAllMembersOf setLE(t2) = False;

    t1 isGTAllMembersOf setGE(t2) = False;
```

```
            t1 gt t3 =>
                t1 isGTAllMembersOf setInterval(t2,t3) = True;
            not(t1 gt t3) =>
                t1 isGTAllMembersOf setInterval(t2,t3) = False;

            t1 isGTAllMembersOf (s1 Union s2)
                = (t1 isGTAllMembersOf s1) and (t1 isGTAllMembersOf s2);


        (*
         * Now define Min, Max
         * for terms constructed from setEQ, setLE, setGE,
         * setInterval and Union
         *)

        ofsort TimeSort

            (*
             * Define Min
             * for setEQ, setLE, setGE, setInterval, Union
             *)
            Min(setEQ(t1)) = t1;
            Min(setLE(t1)) = 0;
            Min(setGE(t1)) = t1;
            Min(setInterval(t1,t2)) = t1;
            Min(s1) le Min(s2)  => Min(s1 Union s2) = Min(s1);
            Min(s1) gt Min(s2)  => Min(s1 Union s2) = Min(s2);

            (*
             * Define Max
             * for setEQ, setLE, setGE, setInterval, Union
             *)
            Max(setEQ(t1)) = t1;
            Max(setLE(t1)) = t1;
            Max(setGE(t1)) = 0; (* For completeness only, this instantiation
                                 * is not used in TLOTOS semantics *)
            Max(setInterval(t1,t2)) = t2;
            Max(s1) ge Max(s2)  => Max(s1 Union s2) = Max(s1);
            Max(s1) lt Max(s2)  => Max(s1 Union s2) = Max(s2);

endtype (* SetGeneratorFunctionsType *)

behaviour

stop

endspec
```

Appendix D

Example application of TLOTOS semantics

This appendix supplies a simple demonstration of the TLOTOS semantics defined in section 6.5.4. We take a simple TLOTOS behaviour expression as our example. We use this to instantiate the appropriate TLOTOS semantic axioms and inference schemas, and hence find the meaning of the expression.

D.1 An example TLOTOS behaviour expression

Consider the following TLOTOS behaviour expression:

$$(a\{5\}; B_1 [] a\{2,3,4\}\mathbf{ASAP}; B_2)|[a]|a\{3,4,5\}; B_3$$

To find the meaning of this expression we use the axioms and schemas defined in section 6.5.4. (For convenience we deviate slightly from the notation used in section 6.5. For instance, we write $\{2,5,\ldots\}$ instead of $\{$*setEQ(2) Union setEQ(5) Union...*$\}$.)

The TLOTOS semantics are defined using [Plo81]'s structured operational approach. To find the meaning of our example TLOTOS behaviour expression we first find the meanings of the component parts of the expression and then the meaning of their composition, using section 6.5.2's axioms and schemas.

D.2 Using axioms for action-prefix-expressions

Our example TLOTOS behaviour expression contains three *action-prefix-expressions*. Their meanings are defined by instantiating section 6.5.4's axioms for *action-prefix-expressions* as shown below.

$$\prec a\{5\}; B_1, 1 \succ -a\{5\}(Normal)(t = 5) \rightarrow$$

We assume that at the initial state of our example behaviour expression the time equals 1.

$-a\{5\}(Normal)(t = 5) \rightarrow$ means that event a can occur at time 5. $\{5\}$ and *Normal* are terms of sort *TimeSetSort* and *NegotiatedTimePolicySort*. As explained in section 6.5.4.4, these two terms are required for the definition of the semantics. They are used to negotiate the outcome of synchronizing event offers.

$$\prec a\{2,3,4\}\mathbf{ASAP}; B_2, 1 \succ -a\{2,3,4\}(Asap)(t = 2) \rightarrow$$

$-a\{2,3,4\}(Asap)(t = 2) \rightarrow$ means that if the above *action-prefix-expression* is considered in isolation then the event a can occur at time 2. However, if the above *action-prefix-expression* was placed in a context which would *Annihilate* the *Asap time-policy* then the event a might occur at any one time in the set $\{2,3,4\}$.

$$\prec a\{3,4,5\}; B_3, 1 \succ -a\{3,4,5\}(Normal)(t \in \{3,4,5\}) \rightarrow$$

$-a\{3,4,5\}(Normal)(t \in \{3,4,5\}) \rightarrow$ means that event a can occur at any one time in the set $\{3,4,5\}$.

D.3 Using schemas for choice-expressions

The meaning of the *choice-expression* in our example, is defined by instantiating section 6.5.4's schemas for *choice-expressions* as follows.

$$\frac{\prec a\{5\}; B_1, 1 \succ -a\{5\}(Normal)(t = 5) \rightarrow}{\prec a\{5\}; B_1 [] a\{2,3,4\}\mathbf{ASAP}; B_2 \succ -a\{5\}(Normal)(t = 5) \rightarrow}$$

$$\frac{a\{2,3,4\}\mathbf{ASAP}; B_1, 1 \succ -a\{2,3,4\}(Asap)(t = 2) \rightarrow}{\prec a\{5\}; B_1 [] a\{2,3,4\}\mathbf{ASAP}; B_2 \succ -a\{2,3,4\}(Asap)(t = 2) \rightarrow}$$

The two schema instances indicate that there are two alternative behaviours for the *choice-expression*: either event a occurs at time 5, or event a occurs at time 2.

D.4 Using schemas for parallel-expressions

At the highest level of composition our example TLOTOS expression is a *parallel-expression*. Its meaning is defined by instantiating section 6.5.4's schemas for *parallel-expressions*. Below, we instantiate these schemas using the results from the axiom and schema instances shown above.

$$\frac{\begin{array}{c}\prec a\{5\}; B_1 [] a\{2,3,4\}\mathbf{ASAP}; B_2 \succ -a\{5\}(Normal)(t = 5) \rightarrow, \\ \prec a\{3,4,5\}; B_3, 1 \succ -a\{3,4,5\}(Normal)(t \in \{3,4,5\}) \rightarrow\end{array}}{\prec (a\{5\}; B_1 [] a\{2,3,4\}\mathbf{ASAP}; B_2) |[a]| a\{3,4,5\}; B_3, 1 \succ -a\{5\}(Normal)(t = 5) \rightarrow}$$

The instantiated schema above says that event a can occur at time 5. This is one possible behaviour of our example TLOTOS behaviour expression.

$$\frac{\prec a\{5\}; B_1 [] a\{2,3,4\}\mathbf{ASAP}; B_2 \succ -a\{2,3,4\}(Asap)(t=2) \rightarrow, \quad \prec a\{3,4,5\}; B_3, 1 \succ -a\{3,4,5\}(Normal)(t \in \{3,4,5\}) \rightarrow}{\prec (a\{5\}; B_1 [] a\{2,3,4\}\mathbf{ASAP}; B_2)|[a]|a\{3,4,5\}; B_3, 1 \succ -a\{3,4\}(Asap)(t=3) \rightarrow}$$

The instantiated schema above says that event a can occur at time 3. This is another possible behaviour of our example TLOTOS behaviour expression.

The above schema negotiated a set of times $\{3,4\}$ ($= \{2,3,4\} \cap \{3,4,5\}$) for the two synchronizing event offers. Also negotiated was the *time-policy Asap* (= *Negotiate(Asap,Normal)*, according to the table in figure 6.16). Applying the negotiated *time-policy Asap* to the negotiated set of times $\{3,4\}$, results in the possible occurrence time 3 for event a.

Hence our example TLOTOS behaviour expression can either perform event a at time 5, or event a at time 3.

Appendix E

Supporting must **timing and** time-policies **in LOTOS**

This appendix contains an example which illustrates difficulties of representing aspects of the semantic mechanisms of TLOTOS in the syntax of LOTOS. This is reference material for section 6.6.2.5.

E.1 The TLOTOS version

```
    a <exp_offer_data1> <time_offer_data1>
       <time_policy_data1> <select_pred_data1>;
    ...
|[a]|
    a <exp_offer_data2> <time_offer_data2>
       <time_policy_data2> <select_pred_data2>;
    ...
```

E.2 The LOTOS version

```
    (
        a_offer !exp_offer_data1 !select_pred_data1
                 !time_offer_data1 !time_policy_data1;
        a_collected;
        a_fired ?exp_offer_data:ExpOfferDataSort;
        ...
    |[a_collected,a_fired]|
        a_offer !exp_offer_data2 !select_pred_data2
                 !time_offer_data2 !time_policy_data2;
        a_collected;
        a_fired ?exp_offer_data:ExpOfferDataSort;
        ...
    )
|[a_offer,a_collected,a_fired]|
    Arbitration[a_offer,a_enabled,a_fired,t](thetime,{},{},Initial)
```

```
where

  process Arbitration[a_offer,a_collected,a_fired,t]
     (thetime:TimeSort,exp_offers:ExpOffersInfoSort,
      time_offers:TimeOffersInfoSort,state_info:StateSort)
     :noexit :=

        (*
         * collect event "a"'s offer data...
         *)
        a_offer ?exp_offer_data:ExpOfferDataSort
                ?select_pred_data:SelectPredDataSort
                ?time_offer_data:TimeOfferDataSort
                ?time_policy_data:TiemPolicyDataSort;
        Arbitration[a_offer,a_collected,a_fired,t](thetime,
                   Record(exp_offer_data,select_pred_data,exp_offers),
                   Record(time_offer_data,time_policy_data,time_offers),
                   Initial)
     []
        (*
         * the event is not yet enabled or firable so let time do not
         * constrain the progress of time
         *)
        t ? newtime [(newtime gt thetime) and (state_info eq Initial)];
        Arbitration[a_offer,a_collected,a_fired,t]
                  (newtime,exp_offers,time_offers,state)
     []
        a_collected;
        (*
         * when a_collected occurs, this means that all event "a"'s offer
         * data has been collected
         *)
        ([(IsExpOffersInfoSatisfiable(exp_offers) eq Yes) and
          (CanTimeOffersInfoBeSatisfied(thetime,time_offers)] ->
             (*
              * the event can be enabled (i.e. its participating processes
              * can synchronise and experiment-offer values which satisfy its
              * selection-predicate can be found)
              * and
              * the event is fireable (i.e. a firing time can be negotiated
              * in respect of the time-offers and time-policies, and this
              * firing time is >= the present time (thetime))
              *)
             Arbitration[a_offer,a_collected,a_fired,t]
                        (newtime,exp_offers,time_offers,Firable)
        )
     []
        ([(state_info eq Firable)] -> and
             (*
              * event "a" can be fired, so block time from progressing
              * beyond the firing time of event "a" (i.e. enforce "must"
              * timing
              *)
             t ? newtime [(newtime gt thetime) and
                         (newtime le ComputeFiringTime(time_offers)];
             Arbitration[a_offer,a_collected,a_fired,t]
                        (newtime,exp_offers,time_offers,state)
        )
     []
        ([(state_info eq Firable) and
          (thetime eq ComputeFiringTime(time_offers))] ->
             (*
              * It is now time to 'fire' event "a"...
              *)
             a_fired !exp_offer_data:ExpOfferDataSort
                    [exp_offer_data eq NegotiateExpOfferValues(exp_offers)];
                Arbitration[a_offer,a_collected,a_fired,t] (newtime,{},{},Inital)
        )
    endproc (* Arbitration *)
```

Appendix F

XL specifications of the X_Service and X_Service_Agent

This appendix lists a series of specifications of the X_Service and X_Service_Agent. The X_Service and X_Service_Agent are abstractions of generic entities found in the CIM-OSA IIS (see section 5.3). Specifications in this series reflect the design of the X_Service and X_Service_Agent at different stages in the development process, or reflect possible alternate design proto types.

These specifications should be read in conjunction with sections 5.5, 6.2, 6.7 and appendix G.4, which guide the reader through example stages of the development X_Service and X_Service_Agent design.

F.1 X_Service TLOTOS specification 1: Xsrv1T

```
(* Specification of a limited X_Service in TLOTOS, focussing on *)
(* timing aspects. *)

process X_Service[X_ACCP] : noexit :=
          X_ACCP ! Req ? data1:DataSort @t1;
          (
              choice data2:DataSort []
                 ([data2 ne Timeout] ->
                  i {setLE(t1+timeout_period)} ASAP;
                  X_ACCP ! Res ! data2;
                  stop)
          []
              i {setEQ(t1+timeout_period+1)} (* timeout *);
              X_ACCP ! Res ! TimeOut;
              stop
          )
endproc (* X_Service *)
```

F.2 X_Service TLOTOS specification 2: Xsrv2T

This specification differs from Xsrv1T only by the fact that (texually) the first **i** event does not have an ASAP *time-policy*.

```
(* Specification of a limited X_Service in TLOTOS, focussing on *)
(* timing aspects. *)

process X_Service[X_ACCP] : noexit :=
          X_ACCP ! Req ? data1:DataSort @t1;
          (
              choice data2:DataSort []
                  ([data2 ne Timeout]->
                   i {setLE(t1+timeout_period)};
                   X_ACCP ! Res ! data2;
                   stop)
          []
              i {setEQ(t1+timeout_period+1)} (* timeout *);
              X_ACCP ! Res ! TimeOut;
              stop
          )
endproc (* X_Service *)
```

F.3 X_Service_Agent TLOTOS specification 1: Xage1T

```
(* Specification of a limited X_Service_Agent in TLOTOS, focussing on *)
(* timing aspects. *)

process X_Service_Agent[X_ACCP,X_AGEP] : noexit :=
          X_ACCP ! Req ? data1:DataSort @t1;
          (
              X_AGEP ! Req ! data1 {setLE(t1+timeout_period)} ASAP;
              X_AGEP ! Res ? data2:DataSort [data2 ne Timeout] {setLE(t1+timeout_period)} ASAP;
              exit(data2)
          [>
              i {setEQ(t1+timeout_period+1)} (* timeout *);
              exit(TimeOut)
          ) >> accept data2:DataSort in
              X_ACCP ! Res ! data2;
              stop
endproc (* X_Service_Agent *)
```

F.4 X_Service_Agent TLOTOS specification 2: Xage2T

This specification is Xage1T sttributed with X_Service_Agent Management functionality.

```
(* Specification of a limited X_Service_Agent extended with *)
(* X_Management functionality, in TLOTOS, focussing on timing aspects. *)

process X_Service_Agent_Ext[X_ACCP,X_AGEP,X_Mgnt] : noexit :=
      X_ACCP ! Req ? data1:DataSort @t1;
      (
          (
              (
                (
                    X_AGEP ! Req ! data1 {setLE(t1+timeout_period)} ASAP;
                    X_AGEP ! Res ? data2:DataSort [data2 ne Timeout]
                                   {setLE(t1+timeout_period)} ASAP;
                    exit(data2)
                [>
                    i {setEQ(t1+timeout_period+1)} (* timeout *);
                    exit(TimeOut)
                )
              >> accept data2:DataSort in
                X_ACCP ! Res ! data2; exit
            )
          [>
            X_Mgnt ! CloseDown ALAP; stop
        )
      >>
        X_Mgnt ! CloseDown; stop
    )
  []
    X_Mgnt ! CloseDown; stop
endproc (* X_Service_Agent_Ext *)
```

F.5 X_Service_Agent TLOTOS specification 3: Xage3T

This specification is structurally more complex than Xage2T, in order to ensure that the *X_Mgnt!CloseDown ALAP* event is not offered after an *X_ACCP!Res* event occurrence; but rather, only the *X_Mgnt!CloseDown* event is offered immediately after the *X_ACCP!Res* event occurrence. (However, section G.4 shows that Xage2T and Xage3T are actually $=_{tc}$.)

```
(* Specification of a limited X_Service_Agent extended with *)
(* X_Management functionality, in TLOTOS, focussing on timing aspects. *)

process X_Service_Agent_Ext[X_ACCP,X_AGEP,X_Mgnt] : noexit :=
      X_ACCP ! Req ? data1:DataSort @t1;
      (
          (
              (
                  X_AGEP ! Req ! data1 {setLE(t1+timeout_period)} ASAP;
                  X_AGEP ! Res ? data2:DataSort [data2 ne Timeout]
                                 {setLE(t1+timeout_period)} ASAP;
                  exit(data2)
              [>
                  i {setEQ(t1+timeout_period+1)} (* timeout *);
                  exit(TimeOut)
              )
            [>
              X_Mgnt ! CloseDown ALAP; stop
          )
        >> accept data2:DataSort in
          (
              X_ACCP ! Res ! data2;
              X_Mgnt ! CloseDown; stop
            []
              X_Mgnt ! CloseDown ALAP; stop
          )
      )
    []
      X_Mgnt ! CloseDown; stop
endproc (* X_Service_Agent_Ext *)
```

BLANK
IN
ORIGINAL

Appendix G

Testing: TLOTOS relations

Chapter 6 established the motivation for TLOTOS and its definition. This appendix takes this work a stage further by proposing and examining useful TLOTOS relations. We define TLOTOS formal relations, such as testing congruence and equivalence, **cred**, **cext** and **red**, demonstrate their application for a few small size, but interesting examples, and then use these relations to test that our CIM OSA SE example specifications, evolved in chapters 5 and 6, are satisfactorily related. First though, to provide us with a perspective on formal relations, we begin with a brief overview of existing Standard LOTOS formal relations.

G.1 Introduction: why we need equivalence (etc.) relations

Most process algebras use *labelled transition systems* (LTSs) as common basis for their semantics.[1] Usually such LTSs are defined by a *structured operational semantics* (SOS) in the style of [Plo81].

Ideally, any two processes which we would want to consider equivalent would have the same LTS as their semantic definition. ([HM85] calls a semantics with this property *fully abstract*). However, we usually find that LTSs are over specifications of process behaviour, in the sense that two processes which we may wish to consider as equivalent for some particular purpose may reduce to distinct LTSs. We therefore choose to consider certain distinct LTSs to describe the same process (or equivalent processes).

A rich web of equivalence relations exists for process algebra. Each particular equivalence relation identifies sets of distinct LTSs which represent processes which are equivalent in some particular sense. Identifications are based on comparisons within a combination of process characteristics, e.g. *traces*, *refusal sets*, *bounded branching*, *copying*, *global testing*, *probabilistic testing*, etc. (see [HM85, Abr87, LS89]).

[1] Usually systems are described in the syntax of process algebra, rather than directly in terms of LTSs, because process algebra syntax provides a convenient, finite means of describing LTSs with huge, if not infinite numbers of states.

This appendix is concerned with defining relations for TLOTOS whose validity can be established through testing.

G.2 Overview of LOTOS relations

A number of formal relations have been defined for comparing LOTOS specifications. Most of these make *identifications* on the basis of observable behaviour. *Stronger* relations make less identifications.

Each relation falls into a number of different categories, such as:

Asymmetric Relations: For terms a, b, and asymmetric relation R:

$a \; R \; b$ does not imply $b \; R \; a$

Symmetric relations: For terms a, b, and symmetric relation $=_R$:

$a =_R b$ *iff* $b =_R a$

Symmetric relations which are associative and transitive are usually called *equivalences*.

Congruence relations: For terms a, b, a symmetric congruence relation $=_{CR}$, and a context $C[.]$:

$a =_{CR} b$ implies $C[a] =_{CR} C[b]$

That is to say, terms identified by an equivalence (using some sense of equality) are equivalent (in the same sense of equality) when *substituted* into a context. In other words, congruences are a subset of equivalences (i.e. they make less identifications), and allow substitutions into all LOTOS contexts.

This section summarizes several of the better known of the LOTOS formal relations. Figure G.1 may help place the relations described in this section in perspective to one another.

The definitions of LOTOS relations vary between authors. This section is intended to provide the reader with a flavour of these relations. See [Abr87, BS86, Mil80, Led91a, Led90, BC89, Pir91] for more detailed treatments of formal LOTOS relations.

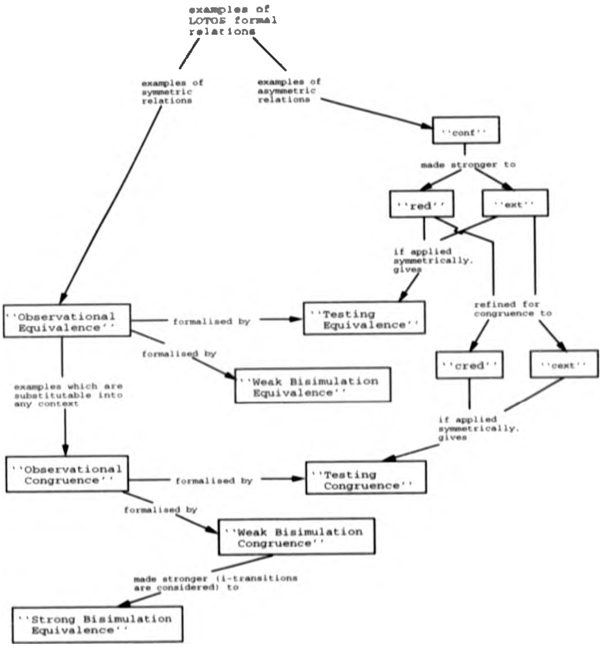

examples of LOTOS formal relations
examples of symmetric relations
examples of asymmetric relations
''conf''
made stronger to
''red''
''ext''
if applied symmetrically, gives
''Observational Equivalence''
formalised by
''Testing Equivalence''
refined for congruence to
formalised by
''cred''
''cext''
''Weak Bisimulation Equivalence''
examples which are substitutable into any context
if applied symmetrically, gives
''Observational Congruence''
formalised by
''Testing Congruence''
formalised by
''Weak Bisimulation Congruence''
made stronger (i-transitions are considered) to
''Strong Bisimulation Equivalence''

Figure G.1: LOTOS relations in perspective

G.2.1 Trace equivalence ($=_{tr}$)

Trace equivalence is a weak equivalence which makes more identifications than are useful for most purposes. We mention it for completeness, and so that it can be compared with the other relations.

Informally, two process are trace equivalent if they can generate equal sets of event sequences (traces).

Consider processes A and B (figure G.2):

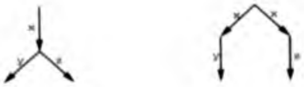


Figure G.2: Processes A and B

Given the definition of trace equivalence:

$$A =_{tr} B$$

where $=_{tr}$ denotes trace equivalence, since:

A generates $\{\prec\succ, \prec x \succ, \prec xy \succ, \prec xz \succ\}$
B generates $\{\prec\succ, \prec x \succ, \prec xy \succ, \prec xz \succ\}$

G.2.2 Observational equivalence

Observational equivalence fulfills the need for an equivalence which distinguishs observable behaviour (e.g. process A and B in figure G.2) but not structural complexity (e.g. processes C and D in figure G.3).

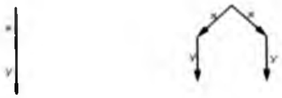


Figure G.3: Processes C and D

G.2.3 Strong bisimilar equivalence ($=_{sbe}$)

Strong bisimilar equivalence is an instance of observable equivalence.

Informally, two process P and Q are strongly bisimilar if the nodes of their behaviour trees[2] are bisimilar. Two nodes P' and Q' are strongly bisimilar *iff* for each event α (observable or internal) offered as a transition from node P' to a node P'', an event α will lead from node Q' to node Q'', and P'' and Q'' are themselves strongly bisimilar, and vice versa.

[2]In this subsection we freely mix tree notation and LOTOS syntax to represent LOTOS processes.

Given this definition:

$$A \neq_{sbe} B$$
$$C =_{sbe} D$$

where $=_{sbe}$ denotes strong bisimilar equivalence (sometimes denoted as $\approx$ by other authors).

G.2.4 Weak bisimilar equivalence ($=_{wbe}$)

Strong bisimilar equivalence makes no distinctions between internal events and observable events. Given our interest in observable behaviour we would like a equivalence which takes into account the special nature of **i** events (e.g. we would like process E and F (figure G.4) to be in some sense equivalent). Weak bisimilar equivalence satisfies this concern.

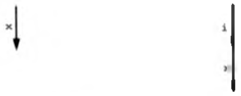


Figure G.4: Processes E and F

Informally, Two process P and Q are weakly bisimilar if the nodes of their behaviour trees are weakly bisimilar. Two nodes P' and Q' are weakly bisimilar *iff* for each event α (observable or internal) offered as a transition from P' to P'', an event sequence Υ from node Q' to Q'' can be found, and P'' and Q'' are themselves weakly bisimilar, and vice versa. Υ is an arbitrarily long sequence of i events with the α event embedded at any point.

Given this definition:

$$E =_{wbe} F \text{ although } E \neq_{sbe} F$$
$$C =_{wbe} D$$

where $=_{wbe}$ denotes weak bisimilar equivalence (sometimes denoted as $\sim$ by other authors).

As a counter example, consider process G and H in figure G.5.

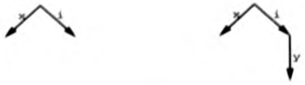


Figure G.5: Processes G and H

$G \neq_{wbe} H$

G.2.5 Weak bisimilar congruence ($=_{wbc}$)

We have established that:

$$E =_{wbe} F$$
$$G \neq_{wbe} H$$

Now notice that:

$$G := C[E]$$
$$H := C[F]$$

where the context $C[.]$ is defined by:

$$C[P] := P [] x; stop$$

This indicates that weak bisimilar equivalent identifications, such as E and F, may be context sensitive (i.e. weak bisimilar equivalence does not necessarily identify congruences). We find that [] and $\rhd$ are the contexts which destroy weak bisimilar equivalence. A stronger form of weak bisimilar equivalence which identifies congruences is known as weak bisimilar congruence.

$$P =_{wbc} Q \text{ if } C[P] =_{wbe} C[Q], \text{ for all contexts } C[.]$$

where $=_{wbc}$ denotes weak bisimilar congruence (sometimes denoted as $\sim_c$ by other authors).

Given this definition,

$$x; x; stop =_{wbc} x; i; x; stop$$
$$x; stop [] i; x; stop =_{wbc} i; x; stop$$

G.2.6 Verification and testing for relations

(Given that our interest lies with investigating testing congruence for TLOTOS, we take this opportunity to introduce some theory on testing by providing definitions of relations using testing theory. Before looking at such relations, the following paragraphs place testing in relation to verification.)

Broadly speaking, there are two means of determining whether a given relation holds between two descriptions: verification and testing. Verification involves *proving* that two descriptions are *equivalent* based on some notion of *equality*. Testing establishes if a more implementation oriented description *conforms* to a more requirements oriented description.

In practice, verification involving "large" descriptions is, at best, complex and resource consuming, at worst practically impossible given current theories and technology. Verification *on-the-fly*[3] using *correctness preserving transformations* provides a possible solution, but identifying *and* formalizing useful general transformations has not proved easy[4]. Testing has reached a state of greater maturity and usefulness.

G.2.7 Testing theory

In testing theory a system is defined by the way in which it responds to tests. A system is, in some sense, *conformant* if it responds appropriately to a particular set of tests.

A typical test composition for a LOTOS specification is:

SpecificationUnderTest[<gates>] ||<gates>|| Test[<gates>, Success]

A test *successfully terminates* for a test execution if a *Success* event is offered. A test unsuccessfully terminates for a test execution if the execution deadlocks without offering a *Success* event.

We summarize the definitions for the testing relations, of [Abr87, BS86, Mil80, Led91a, Led90, BC89, Pir91] in the remainder of this section.

LOTOS may describe systems which exhibit non deterministic behaviour. Therefore, two basic types of test response have been defined: *may response* and *must response*.

May response: A test *T* has a *may response* when applied to a LOTOS specification *S* if it *successfully terminates* for at least one execution of the test composition.

Must response: A test *T* has a *must response* when applied to a LOTOS specification *S* if it *successfully terminates* for every execution of the test composition.

Also, we identify two basic types of tests: *may tests* and *must tests*.

May test: A *may test* of *S*, written *May(S)*, is a test which gives a *may response* when applied to *S*.

Must test: A *must test* of *S*, written *Must(S)*, is a test which gives a *must response* when applied to *S*.

We can use specific types of tests to check particular properties of specifications. Examples of these tests are: the *may sequential test*, the *refusal set test*, and the *existential refusal set test*.

The may sequential test: A *may sequential test* of a specification *S*, is a sequential test which has a may response when applied to *S*.

The *may sequential test* can be used to determine possible traces of a specification. (Trivially, a sequential test may contain no observable transitions.) The following template characterizes a sequential test:

[3] As opposed to *post* verification discussed in the previous sentence.

[4] Hardly surprising since this task appears similar to coding and automating design creativity.

```
process SequentialTest [<gates>, Success] : noexit :=
    <event1>;
        ...

    <eventn>;
    Success;
    stop
endproc (* SequentialTest *)
```

The refusal set test: The *refusal set test* can be used to check that no events from a particular set are offered at the system state where the test is applied. The following template characterizes a refusal set test:

```
process RefusalSetTest [<gates>, Success] : noexit :=
        <rejected_event1>; stop
    []
                :
                :
    []
        <rejected_eventn>; stop
    []
        Success; stop
endproc (* refusalSetTest *)
```

The existential refusal set test: The *existential refusal set test* (ERS test) can be used to check *refusal sets after a given observable transition.*

An ERS test is a composition of a may sequential test and a refusal set test. An ERS test successfully terminates if the *may sequential test* leads to a state where the application of the *refusal set test* has a *must* termination.

The following template characterizes an ERS test:

```
process ERS_Test [<gates>, Success1, Success2] : noexit :=
    <accepted_event1>;
            :
            :
    <accepted_eventn>;
    Success1;
    (
        <rejected_event1>; stop
    []
                :
                :
    []
        Success2; stop
    )
endproc (* ERS_Test *)
```

G.2.8 Testing equivalence ($=_{te}$)

This is an interesting equivalence because it is a slightly weaker form of weak bisimulation equivalence.

Two specifications S_1 and S_2 are testing equivalent if every *may* and *must* test of S_1 is also a *may* and a *must* test respectively of S_2, and vice versa.

Testing equivalence cannot distinguish between specifications that cannot be distinguished by experiments, while weak bisimulation equivalence may make such distinctions. This point is illustrated by the following example.

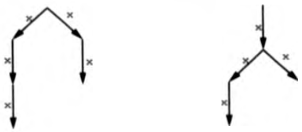


Figure G.6: Processes I and J

For process I and J (figure G.6):

$I =_{te} J$ but $I \neq_{wbe} J$

G.2.9 Asymmetric relations

Leaving equivalences aside for the moment, we survey a number of asymmetric relations which have been proposed. These are interesting because they reflect the asymmetric character of the development process, where a specification S_1 in some sense "implements" a description S_2 but the opposite is not true. See [BS86, Led91a, Led90] for detailed insights into this issue.

G.2.10 The conformance relation (conf)

S_1 **conf** S_2 iff (for every ERS test formed as described below) there exists (a *must* run of) an ERS test applied to S_1 then there exists (a *must* run of) the same ERS test applied to S_2. An ERS test of S_1 is formed from a *may sequential test* of S_2 followed by a *refusal set test* formed from the union of events in S_1 and S_2.

conf is not transitive. Hence it is possible that:

K **conf** L **conf** M, but K $\cancel{\textbf{conf}}$ M

where K, L and M are the processes in figure G.7.

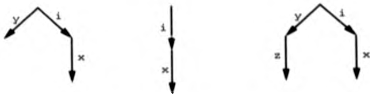


Figure G.7: Processes K, L and M

To understand why K ~~conf~~ M, consider the following ERS test:

```
process ERS_Test1 [x,y,z,Success1,Success2] : noexit :=
    y; Success1; (z; stop [] Success2; stop)
endproc (* ERS_Test1 *)
```

ERS_Test1 is an ERS test of K, in the sense described in the definition above. In composition with K there is a test run instance which must generate the trace $\prec y, Success1, Success2 \succ$. In composition with M there exists no test run instance which must generate this trace. Therefore K ~~conf~~ M by the above definition.

G.2.11 The reduction relation (red and cred)

S_1 **red** S_2, if S_1 **conf** S_2 and the trace set of S_1 is a subset of the trace set of S_2. **cred** is the subset of congruent **red** relations.

The reduction relation formalizes the notion of a reduction of non-determinism. The behaviour of an *implementation* is an acceptable reduction of the behaviour of the specification.

Example:

L **red** M

Also, consider the following example (figure G.8) taken from [BB87].

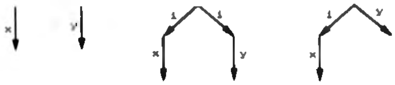


Figure G.8: Processes N, O, P and Q

$$N \ \mathbf{red} \ P \tag{G.1}$$
$$N \ \mathbf{red} \ Q \tag{G.2}$$
$$O \ \mathbf{red} \ P \tag{G.3}$$
$$O \ \cancel{\mathbf{red}} \ Q \tag{G.4}$$

The reason why **red** holds in equation G.2, but not in equation G.4, is quite subtle. To understand this reason consider the following two ERS tests:

```
process ERS_Test_NandQ [x,y,Success1,Success2] : noexit :=
   Success1; (y; stop [] Success2; stop)
endproc (* ERS_Test_NandQ *)

process ERS_Test_OandQ [x,y,Success1,Success2] : noexit :=
   Success1; (x; stop [] Success2; stop)
endproc (* ERS_Test_OandQ *)
```

(Notice, that in both of the above ERS tests, there are no observable events before the *Success1* event. This trivial ***sequential test*** allows for the cases where there are no observable transitions before the ***refusal set test***.)

ERS_Test_NandQ is an ERS test of N in the sense described in the definition above. In composition with N it has a test run instance which must generate the trace $\prec Success1, Success2 \succ$. Also, in composition with Q it has a test run instance which must generate this trace. It follows that N **conf** Q.

On the other hand: *ERS_Test_OandQ* is an ERS test of O in the sense described in the definition above. In composition with O it has a test run instance which must generate the trace $\prec Success1, Success2 \succ$. However, in composition with Q there exists no test run instance which must generate this trace. Therefore O $\not\mathbf{conf}$ Q.

G.2.12 The extension relation (ext and cext)

S_1 **ext** S_2 if S_1 **conf** S_2 and the trace set of S_2 is a subset of the trace set of S_1. **cext** is the subset of congruent **ext** relations.

The extension relation formalizes the notion of preserving and extending the functionality of the ***specification*** in the ***implementation*** in a controlled manner (any sequence of events accepted by the ***specification*** will also be accepted by the ***implementation***).

Example:

K **ext** L

G.3 Testing relations for TLOTOS

We use the review in the previous section of LOTOS formal relations as a basis for investigating TLOTOS formal relations. We consider only those relations which can be defined by testing, having dismissed in subsection G.2.6 the current verification methods as too expensive or impossible to implement.

We begin by immediately considering ***testing congruence*** for TLOTOS.

G.3.1 TLOTOS testing congruence ($=_{tc}$)

We initially assume that TLOTOS $=_{tc}$ is defined similarly to LOTOS $=_{tc}$, i.e. that:

$$S_1 =_{tc} S_2 \Leftrightarrow \quad (Must(T[S_1]) \Leftrightarrow Must(T[S_2])$$
$$\wedge \quad May(T[S_1]) \Leftrightarrow May(T[S_2])) \text{ for all test contexts } T[.]$$

Now we investigate if this definition of TLOTOS $=_{tc}$ is useful.

G.3.1.1 Case 1

Given:

S$_1$:= a; **stop** S$_2$:= a **ASAP**; **stop**

Is $S_1 =_{tc} S_2$? We define[5] the following test context $C[.]$:

. |[a]| a; Success {0}; **stop**

Using this test context in a *must test*, we find that $C[S_1]$ fails (i.e. a *Success* event does not occur for every test run), whereas $C[S_2]$ succeeds (i.e. a *Success* event does occur for every test run). Therefore:

$S_1 \neq_{tc} S_2$

This result confirms our intuition about how S_1 compares to S_2. The test context makes use of the fact that the initial state of all TLOTOS specifications is given the time value 0 (see section 6.5.4.1). In S_2, the a event is specified to occur ASAP, the initial time is 0, and there are no other constraints preventing the occurrence of event a. Therefore for S_2, a must occur at time 0. In contrast, S_1 specifies no ASAP urgency, and event a may occur at any time ≥ 0.

For testing with TLOTOS, we (unsurprisingly) specify occurrence times for events, including the *Success* event, within the test context process. This allows us to differentiate specifications that are identical in terms of relative ordering, but which differ in the occurrence times of their respective events. Also, the test context can be combined with the specification under test using a selection of any of the TLOTOS behaviour operators. For instance, to test an **exiting** TLOTOS specification, the test context may be combined as follows:

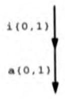
```
(
   SpecificationUnderTest
|[<gates>]|
   TestPart1[<gates>]
)
>> (* TestPart2 *) Success {setLE(8)}; stop
```

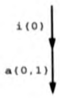
This example can be used to test that the *SpecificationUnderTest may* **exit** within 8 units of time.

[5]In the remainder of this chapter, for convenience, we write '{3,7,...,9}' instead of '*{setEQ(3) Union setEQ(7) Union ... Union setEQ(9)}*', etc.

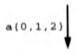
G.3.1.2 Cases 2-6

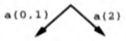
There follows a selection of small, but interesting TLOTOS cases for testing congruence.



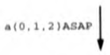


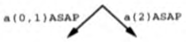
i {0,1}; a {0,1}; **stop** $\neq_{tc}$ i {0}; a {0,1}; **stop**



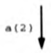


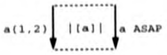
a {0,1}; **stop** []a {2}; **stop** $=_{tc}$ a {0,1,2}; **stop**



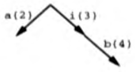


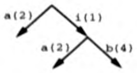
a {0,1} **ASAP**; **stop** []a {2} **ASAP**; **stop** $\neq_{tc}$ a {0,1,2} **ASAP**; **stop**





a {2}; **stop** $\neq_{tc}$ a {1,2}; **stop** |[a]| a **ASAP**; **stop**





a {2}; **stop** []i {3}; b {4}; **stop** $=_{tc}$ a {2}; **stop** []i {1}; (a {2}; **stop** []b {4}; **stop**)

G.3.1.3 Assessment of TLOTOS $=_{tc}$

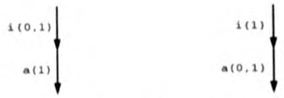
Using the definition of TLOTOS testing congruence, as given at the start of this subsection, we have considered a number of example cases (including those above). Studying these cases, we believe this definition of TLOTOS testing congruence to make useful, intuitive identifications.

G.3.2 TLOTOS cred and cext relations

We now consider the TLOTOS asymmetric relations **cred** and **cext**. We initially assume that the TLOTOS versions of **cred** and **cext** are defined similarly to the LOTOS versions of these relations (see subsection G.2.9).

As done for TLOTOS $=_{tc}$, we now investigate whether these definitions for TLOTOS **cred** and **cext** are useful.

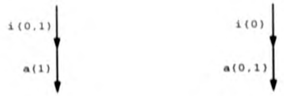
G.3.2.1 Case 1



i {0,1}; a {1}; **stop** **cred** i {1}; a {0,1}; **stop**

This is a trivial case.

G.3.2.2 Case 2



S_1 :=i {0,1}; a {1}; **stop** S_2 :=i {0,1}; a {0,1}; **stop**

Conjecture:

$S_1 \ \mathbf{cr\not{e}d} \ S_2$

The prove this case, we prove that:

$C[S_1] \ \mathbf{re\not{d}} \ C[S_{2}.$

Consider the $C[.]$ as the context:

(. |[a]| a **ASAP**; **stop**)

And consider the following ERS test:

```
process ERS_Test_cred2 [a,Success1,Success2] : noexit :=
    Success1; (a{0}; stop [] Success2; stop)
endproc (* ERS_Test_cred2 *)
```

Now since, $C[S_1]|[a]|ERS\_Test\_cred2$ *must* generate the trace $\prec Success1, Success2 \succ$, but $C[S_2]|[a]|ERS\_Test\_cred2$ only *may* generate this success trace, it follows that $S_1 \ \not\mathbf{conf}\ S_2$, and hence that $S_1 \ \not\mathbf{cred}\ S_2$.

G.3.2.3 Case 3

a(0,1) ASAP
a(0,1)

S_1 := a {0,1} **ASAP**; **stop** S_2 := a {0,1}; **stop**

Conjecture:

$$S_1 \quad \mathbf{cred} \quad S_2$$

$C[S_1]$ and $C[S_2]$ both produce the same responses to ERS tests of S_1, hence $S_1 \mathbf{conf} S_2$. Also, since the trace set of S_1 in any context $C[.]$ is a subset of the trace set of S_2 in the same context (i.e. $Tr(C[S_1]) \subseteq Tr(C[S_2])$), our conjecture is proved correct.

S_1 and S_2 can be distinguished by a suitable set of *must* and *may tests*, which implies that are they are not testing congruent. This is confirmed by the fact that $Tr(C[S_2]) \not\subseteq Tr(C[S_1])$, hence $S_2 \ \not\mathbf{cred}\ S_1$ and therefore $S_1 \neq_{tc} S_2$.

G.3.2.4 Case 4

Conjecture:

S_2 **cext** S_1 (for S_1, S_2 as in case 3, above)

Proof:

$$\begin{array}{rcl} C[S_2] & \mathbf{conf} & C[S_1] \\ \wedge\ Tr(C[S_2]) & \supseteq & Tr(C[S_1]) \end{array}$$

G.3.2.5 Assessment of TLOTOS cred and cext

We have demonstrated that **conf**, **cred** and **cext** make sensible, intuitive identifications for TLOTOS.

G.3.3 Non-congruent relations for TLOTOS

The previous subsections have concentrated on the congruence relations $=_{tc}$, **cred** and **cext**, and have not examined $=_{te}$, **red** and **ext** in their own right. These latter relations can be considered as congruences where the context variable $C[.]$ is constrained to be the particular process that is cabable of both doing and refusing any event. The non-congruent relations also yield sensible results when applied to TLOTOS descriptions.

G.4 Testing CIM-OSA specifications

In chapters 5 and 6 we saw how we could use TLOTOS to describe CIM-OSA SE specifications, and how the TLOTOS descriptions overcame the inadequacies of the LOTOS descriptions introduced in chapter 6. Having investigated formal relations using small TLOTOS specifications, we now employ these same relations to demonstrate how to check that our CIM-OSA SE TLOTOS specifications (appendix F) meet the requirements discussed in sections 6.2, 6.7 and 5.5.

G.4.1 Our approach

A number of discussions and theories have been documented, and tools developed in the field of (automatic) test suite generation and application for LOTOS. In particular, sources such as [BS86, Wez89, Led91a, Mil80] have been the inspiration, and projects such as SEDOS, PANGLOSS and LOTOSPHERE have been the genesis of tools such as SQIGGLES [BC89], LOLA [QPF89], HIPPO [Mar89], TOPO [MdM89], COOPER [Ald90], etc. Such tools contribute towards an automated testing process.

The development and use of such tools for automating TLOTOS tests is outside the scope of this thesis. We are interested only in investigating whether the testing theories for TLOTOS are useful, and can be applied. For the testing examples documented in this section, we generate, apply and analyse the tests manually. No attempt is made to automate this process.

G.4.2 The specification subjects

The specifications, informally described in section 6.2, and formally described in section 6.7 and appendix F, are abstractions of the S_1, S_2 and S_3 specifications discussed in section 5.5. In this section we explore the web formal relations between these (abstract) specifications. We might consider this web as a map through aspects of the development process, in a way similar to figure 5.16 (see the discussion in section 5.5). Also, we answer the example questions raised at the end of section 6.7.

G.4.3 Relations between the SE_Service and the SE_Service_Agent

Appendixes F.1 and F.3 contain (abstractions of) specifications of the SE_Service and SE_Service_Agent. Now, we provide an example of the process of investigating what relations hold between these specifications: we make an example conjecture and test its truth.

G.4.3.1 Xage1T cconf Xerv1T

We conjecture that the SE_Service_Agent specification *Xage1T* (in any context $C[.]$) conforms to the SE_Service specification *Xerv1T* (in the same context $C[.]$). Stated another way, we would like the SE_Service_Agent specification to preserve the func-

tionality of the SE_Service specification. To prove our conjecture we test the relation: $C[Xage1T]$ **conf** $C[Xsrv1T]$, where $C[.]$ is any context.
Consider $C[.]$ as the context[6]:

```
(
            ( )
|[X_AGEP]|
            (X_AGEP ! Req ? data1:DataSort ALAP;
             X_AGEP ! Res ? data2:DataSort ALAP;
             stop)
)
```

and consider the following ERS test:

```
process ERS_Test_X1[X_ACCP,Success1,Success2] noexit :=
   X_ACCP ! Req ? data1 DataSort @t1;
   Success1 ASAP;
   (
      X_ACCP ! Res ? data2 DataSort {setEQ(t1)}; stop
   []
      Success2; stop
   )
endproc (* ERS_Test_X1 *)
```

Now since, $C[Xage1T]|[X\_ACCP]|ERS\_Test\_X1$ does yield *must* (*Success2*) test runs, but $C[Xsrv1T]|[X\_ACCP]|ERS\_Test\_X1$ yields only *may* (*Success2*) test runs, it follows that $Xage1T$ **cconf** $Xsrv1T$.[7]

Thus we have proved that *Xage1T* does not preserve the functionality of *Xsrv1T*. However, we would like this to be the case, so how should we redesign *Xage1T* or *Xsrv1T*? The answer is related to the question raised in the first point at the end of section 6.7. In section 6.7 we doubted if the informal requirements for the X_Service (section 6.2) was a suitable place in which to express the constraint that 'the X_Service should compute and offer an *X_ACCP!Res!data2* event ASAP'. The *Xsrv1T* directly reflects this constraint by placing an ASAP *time-policy* on the i event[8], which represents the computation in question.

As suggested in the first point at the end of section 6.7, we resolve this 'error in the informal requirements' by relegating the 'urgency of computation' constraint from the X_Service requirements, to the X_Service_Agent requirements only. This is reflected in the formal TLOTOS description of the X_Service, through the removal of the ASAP

[6]Notice how we purposely annihilate the ASAP urgency on the *X_AGEP* events of *Xage1T* by placing ALAP *time-policies* on the *X_AGEP* event offers of the context $C[.]$. Annihilation of the ASAP urgency allows *X_AGEP* events to occur at any time in the range $t1 \ldots t1 +$ *timeout_period*, thus simulating the range of occurrence times of *X_AGEP* events if the X_Service_Agent were placed in a 'real' IIS context

[7]**cconf** denotes **conf** congruence.

[8]Textually, the first i event in the *Xsrv1T* specification

time-policy on the i event. This results in the new X_Service specification *Xsrv2T*, in appendix F.2.

Xage1T **cconf** *Xsrv2T* does hold, with the *Xage1T* specification alone reflecting the 'urgency of computation' constraint.

G.4.4 The relations between the Extended SE_Service_Agent and the SE_Service_Agent

Appendixes F.4 and F.5 contain (abstractions of) specifications of the SE_Service_Agent (*Xage2T*) and the Extended SE_Service_Agent (*Xage3T*). Now, we provide an example of the process of investigating what relations hold between these specifications: we make an example conjecture and test its truth.

G.4.4.1 Xage3T $=_{te}$ Xage2T

We conjecture that the extended[9] SE_Service_Agent specification *Xage2T* (in the particular context $C[.]$, where no X_Management functions may occur), is *testing equivalent* to the SE_Service_Agent specification *Xage1T* (in the same context $C[.]$). Stated another way, we would like the extended SE_Service_Agent specification to behave equivalently to the SE_Service_Agent specification, when they are both placed in the context ($C[.]$) where no X_Management functions may occur. To prove our conjecture, we test the relation: $C[Xage1T] =_{te} C[Xage2T]$, where $C[.]$ is the particular context[10]:

```
(
              (.)
|[X_Mgnt]|
              X_Mgnt ! CloseDown {Empty}; stop
)
```

It is trivial to see that:

$$(Must(T[C[Xage1T]]) \Leftrightarrow Must(T[C[Xage2T]])$$
$$\wedge \quad May(T[C[Xage1T]]) \Leftrightarrow May(T[C[Xage2T]])) \text{ for all test contexts } T[.]$$

and hence that $C[Xage1T] =_{te} C[Xage2T]$.

G.4.5 The relation between the Extended SE_Service_Agent specifications: Xage2T and Xage3T

In the last point in section 6.7, we aired a worry that the ALAP *Closedown* event in *Xage2T* could occur immediately after the *X_ACCP!Res!data2* event. Does this

[9]The extended SE_Service_Agent specification *Xage2T* extends the original functionality of the SE_Service_Agent specification *Xage1T* with the X_Management function *CloseDown*.

[10]The context $C[.]$ ensures that the X_Management *CloseDown* function cannot occur, by 'never' (i.e. the *Empty* set) offering to synchronize on the *CloseDown* event.

violate the informal requirements of section 6.2 which indicate that the *Closedown* event should not be required to occur ALAP once the *X_ACCP!Res!data2* event occurs? Specification *Xage3T* represents a restructuring of specification *Xage2T*, written to explicitly avoid this worry. But are *Xage2T* and *Xage3T* different (by testing)? Stated more formally, does the relation $Xage2T \neq_{tc} Xage3T$ hold?

We can capture the essential difference between the two specifications *Xage2T* and *Xage3T* in the abstractions (respectively, *AXage2T* and *AXage3T*):

```
AXage2T := (X_ACCP!Res!data2; exit ▷X_Mgnt!CloseDown; stop) ≫X_Mgnt!CloseDown; stop
AXage3T := (X_ACCP!Res!data2; exit []X_Mgnt!CloseDown; stop) ≫X_Mgnt!CloseDown; stop
```

Testing, we find that $AXage2T =_{tc} AXage3T$. For example, if we take the simple context ($C[.]$):

```
                          (.)
 |[X_ACCP, X_Mgnt]|
                          (X_ACCP!Res!data2 {0}; stop ||| X_Mgnt!CloseDown {0,1}; stop)
```

we find that both $C[AXage2T]$ and $C[AXage3T]$ have the same set of *may tests*:

$$\{\ X\_ACCP!Res!data2_0 \rightarrow X\_Mgnt!CloseDown_0,\ X\_ACCP!Res!data2_0 \rightarrow X\_Mgnt!CloseDown_1,\ X\_Mgnt!CloseDown_1\ \}$$

and the same set of *must tests*:

$$\{\ \}$$

With $AXage2T =_{tc} AXage3T$, it follows that $Xage2T =_{tc} Xage3T$.

Given the equivalence between *AXage2T* and *AXage3T*, a reasonable next step might be to attempt to rationalize either *AXage2T* or *Axage3T*, to:

```
AXage4T := X_ACCP!Res!data2; stop ▷X_Mgnt!CloseDown; stop
```

but the set of *may tests* of $C[AXage4T]$ contains the trace: $X\_Mgnt!CloseDown_0$, and hence $AXage4T \neq_{tc} AXage2T$.

Appendix H

Example application of the SimChar algorithm

This appendix provides an example of the application of the *SimChar* algorithm (section 7.4.7) to a simple PbLOTOS system. This example illustrate how *SimChar* probabilizes an NP-LTS which contains one of the non-deterministic branching scenarios described in section 7.4.3. The appendix provides a a step-by-step guide through the instantiations of *SimChar*, illustrating how *SimChar* produces the set of simultaneous equations which characterize an NP-LTS as a P-LTS. Important points concerning the algorithm's method are highlighted.

H.1 The example PbLOTOS system

The example system (*spec*) to which we apply *SimChar*, is defined by the following PbLOTOS fragment, and represented graphically in figure H.1.

a; **stop** [] i; b; **stop**

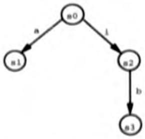


Figure H.1: The PbLOTOS system

The following sections step through the workings of *SimChar* applied to the example NP-LTS above.

H.2 Instantiation for state s_0

The result of the $SimChar(s_0, \ldots)$ instantiation ($SimChar$ applied to the initial state s_0 of the PbLOTOS system $spec$) is described below.

```
$SimChar(s_0, 0, 1, 0, \emptyset)$ is

    (* first create the subset of auxiliary equations associated with state $s_0$ *)
    $auxeq' = \{\ 0 < \mu_{0\ frac} < 1,\ \mu_{0.o.a.1} = \mu_{0\ frac},$
                $0 < \mu_{0.i.1} < 1,\ 0 < \mu_{0.o.a.1} < 1,$
                $\mu_{0\ frac} + \mu_{0.i.1} = 1$
              $\}$

    (* now launch SimChar instantiations to process the states that follow state
     * $s_0$, and unify the trprobs and auxeqs returned from these
     * instantiations when they return *)

    $\prec trprob', auxeq'' \succ :=$
           $SimChar(s_2, 0.i.1, 1, 0, trprob'')$
         $\uplus$
           $SimChar(s_1, 0.o.a.1, 1, 1, trprob''')$

where

    $trprob'' = \{\prec s_0 = \epsilon \Rightarrow s_2, 1 \times (\mu_{0.i.1} + \mu_{0\ frac}) \succ\}$
    $trprob''' = \{\prec s_0 = a \Rightarrow s_1, 1 \times \mu_{0.o.a.1} \succ\}$

    (* and finally, return the unified trace probability sets (trprob'), and
     * return the auxiliary equations (auxeq') for the state $s_0$ concatenated with the
     * auxiliary equations (auxeq'') for the states which follow $s_0$ *)
    $return(\prec trprob', auxeq' \oplus auxeq'' \succ)$
```

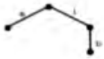
$SimChar(s_2, \ldots)$ processes states which follow the **i** transition from s_0.
$SimChar(s_1, \ldots)$ processes states which follow the a transition from s_0.

We take at look at these $SimChar$ instantiations in a moment, but first we have an important point to make about how $SimChar$ processes state s_0 of this example.

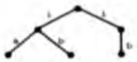
H.3 Discussion

s_0 is an example of a state where non-determinism occurs as a result of a combination of an observable transition an unobservable (**i**) transition (the case $|AllProbPairs(s_k, a)| > 1$ in sections 7.4.3 and 7.3.2).

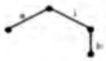
For this case $SimChar$ must preserve the $P(= a \Rightarrow) = 1$ property of the NP-LTS $spec$ in the resulting set of simultaneous equations. To do this, $SimChar$ contains knowledge of the fact that:



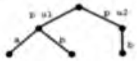
is observationally equivalent to:



This means that the NP-LTS:



can effectively be replaced by the P-LTS:

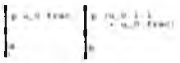


where:

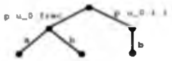
$0 < \mu_1 < 1$
$0 < \mu_2 < 1$
$\mu_1 + \mu_2 = 1$

However, rather than have *SimChar* actually edit the branching structure of the NP-LTS to produce the branching structure of the P-LTS (as shown above), we instead define *SimChar* to generate traces of a P-LTS which has the above branching structure. The difference is not just conceptual. The definition of the *SimChar* algorithm would have been much more complex if we had defined *SimChar* so that, when encountering a non-deterministic state similar to that shown above, it re-arranged the branching structure of the system and then proceeded to act over the new branching structure. Our chosen definition of *SimChar*, which produces traces *as if* the *offending* branching structure had been re-arranged, is much more tidy.

When encountering the non-deterministic scenario at state s_0, *SimChar* allocates the free-term $\mu_{0,frac}$ to be the sum of probabilities of all observable transitions from state s_0 (a only, for this example). *SimChar* allocates 1 to the sum of probabilities of unobservable i transitions from state s_0 (only one i transition for this example). Hence, *SimChar* generates the traces (depicted as branches):



which could be assembled to produce the P-LTS:



(The definition of an P-LTS section ?? tells us that the assembly of the above traces as a P-LTS would have to be as shown since the P-LTS definition disallows any p transition to carry a probability value > 1 (which the second trace does, since $\mu_{0,i\,1} + \mu_{0,frac} = 1$).)

H.4 Instantiation for state s_1

The result of the *SimChar*$(s_1, \ldots)$ instantiation is described below.

```
SimChar(s1, 0, a.a, 1, 1, 1, trprob) is

    return(< trprob, ∅ >)
```

There are no transitions from s_1 (and, anyway, the maximum specified observable trace depth has been reached) so simply return the trace probabilities (*trprob*) and auxiliary equations (*auxeq*) that have been accumulated for this branch.

H.5 Instantiation for state s_2

The result of the $SimChar(s_2, \ldots)$ instantiation is described below.

```
$SimChar(s_2, 0.i.1, 1, 0, trprob, auxeq)$ is

    (* first create the subset of auxiliary equations associated with state $s_2$ ... *)
    $auxeq' = \{\ \mu_{0.i.1.frac} = 1$, (* no unobserv. trans. from $s_2$ so set $\mu_{0.i.1.frac}$ to 0 *)
                $\mu_{0.i.1.o.b.1} = \mu_{0.i.1.frac}$
              }

    (* now launch SimChar instantiation to process the state that follows state
     * $s_2$, and collect the trprob and auxeq sets returned *)

    $\prec trprob', auxeq'' \succ :=$
        $SimChar(s_3, 0.i.1.o.b.1, 1, 1, trprob'')$

where

    $trprob'' = \{\prec s_0 = \epsilon = b \Rightarrow s_3, 1 \times (\mu_{0.i.1} + \mu_{0.frac}) \times \mu_{0.i.1.o.b.1} \succ\}$
              $= \{\prec s_0 = b \Rightarrow s_3, (\mu_{0.i.1} + \mu_{0.frac}) \times \mu_{0.i.1.o.b.1} \succ\}$

    (* and finally return *)
    $return(\prec trprob', auxeq' \oplus auxeq'' \succ)$
```

$SimChar(s_3, \ldots)$ processes states which follow the b transition from s_2.

H.6 Instantiation for state s_3

The result of the $SimChar(s_3, \ldots)$ instantiation is described below.

```
$SimChar(s_3, 0.i.1.o.b.1, 1, 1, trprob, auxeq)$ is

    $return(\prec trprob, \emptyset \succ)$
```

There are no transitions from s_3 (and, anyway, the maximum specified observable trace depth has been reached) so simply return the trace probabilities (*trprob*) and auxiliary equations (*auxeq*) that have been accumulated for this branch.

H.7 Unifying *trprob* sets and *auxeq* sets when rewinding *SimChar* recursion

Once *SimChar* has recursively re-instantiated itself in order to trace all branches of the NP-LTS to which it was applied (*spec* for the example in this appendix), it will have accumulated all the information it needs in a set of pairs, each pair consisting of a $\boldsymbol{trprob}$ set and a $\boldsymbol{auxeq}$ set. Each instantiation returns one of these pairs when it terminates. When rewinding recursive instantiations, the pairs are unified by $\overset{\oplus}{\uplus}$. Returning to the instantiation $SimChar(s_0, \ldots)$, the launched instantiations:

$$\begin{array}{l} \quad SimChar(s_2, 0.i.1, 1, 0, \boldsymbol{trprob''}) \\ \overset{\oplus}{\uplus} \\ \quad SimChar(s_1, 0.o.a.1, 1, 1, \boldsymbol{trprob'''}) \end{array}$$

will return with:

$$\begin{array}{l} \qquad \prec \{\prec s_0 = b \Rightarrow s_3, (\mu_{0.i.1} + \mu_{0.frac}) \times \mu_{0.i.1.o.b.1} \succ\}, \\ \qquad\quad \{\mu_{0.i.1.frac} = 1,\ \mu_{0.i.1.o.b.1} = \mu_{0.i.1.frac}\} \\ \qquad \succ \\ \quad \overset{\oplus}{\uplus} \\ \qquad \prec \{\prec s_0 = a \Rightarrow s_1, 1 \times \mu_{0.o.a.1} \succ\}, \\ \qquad\quad \emptyset \\ \qquad \succ \\ = \\ \qquad \prec \{\prec s_0 = b \Rightarrow s_3, (\mu_{0.i.1} + \mu_{0.frac}) \times \mu_{0.i.1.o.b.1} \succ, \prec s_0 = a \Rightarrow s_1, 1 \times \mu_{0.o.a.1} \succ\}, \\ \qquad\quad \{\mu_{0.i.1.frac} = 1,\ \mu_{0.i.1.o.b.1} = \mu_{0.i.1.frac}\} \\ \qquad \succ \end{array}$$

Finally, the instantiation $SimChar(s_0, \ldots)$ takes the $\boldsymbol{auxeq}$ member of this $\prec \boldsymbol{trprob}, \boldsymbol{auxeq} \succ$ pair and concatenates it with the $\boldsymbol{auxeq}$ it has generated specifically for state s_0, i.e. the $\boldsymbol{auxeq}$:

$$\begin{array}{l} \{\ 0 < \mu_{0.frac} < 1,\ \mu_{0.o.a.1} = \mu_{0.frac}, \\ \quad 0 < \mu_{0.i.1} < 1,\ 0 < \mu_{0.o.a.1} < 1, \\ \quad \mu_{0.frac} + \mu_{0.i.1} = 1 \\ \} \end{array}$$

to compile and return the $\prec \boldsymbol{trprob}, \boldsymbol{auxeq} \succ$ pair result:

$$\begin{array}{l} \prec \{\prec s_0 = b \Rightarrow s_3, (\mu_{0.i.1} + \mu_{0.frac}) \times \mu_{0.i.1.o.b.1} \succ, \prec s_0 = a \Rightarrow s_1, 1 \times \mu_{0.o.a.1} \succ\}, \\ \quad \{\ 0 < \mu_{0.frac} < 1,\ \mu_{0.o.a.1} = \mu_{0.frac}, \\ \qquad 0 < \mu_{0.i.1} < 1,\ 0 < \mu_{0.o.a.1} < 1, \\ \qquad \mu_{0.frac} + \mu_{0.i.1} = 1, \\ \qquad \mu_{0.i.1.frac} = 1,\ \mu_{0.i.1.o.b.1} = \mu_{0.i.1.frac} \\ \quad \} \\ \succ \end{array}$$

Appendix I

Abbreviations

This appendix contains a list of acronyms, initializations and other abbreviations used in the thesis.

Abbreviation	Expansion	Context
ANSA	Advanced Networked Systems Architecture	
AC	Activity Control Service	CIM-OSA
AF	Application Front-End Services	CIM-OSA
ACCP	Access-Protocol	CIM-OSA
AGEP	Agent-Protocol	CIM-OSA
APM	Architecture Projects Management Ltd.	ANSA
B	Business Complex	CIM-OSA
BC	Business Process Control Service	CIM-OSA
C	Communications Complex	CIM-OSA
CCITT	International Consultative Committee on Telegraphy and Telephony	
CIM	Computer Integrated Manufacturing	
CIM-OSA	Computer Integrated Manufacturing – Open Systems Architecture	ESPRIT
CM	Communications Management Service	CIM-OSA
DAF	Distributed Applications Framework	
DM	Data Management Service	CIM-OSA

Abbreviation	Expansion	Context
ESPRIT	European Strategic Programme for Research and Development in Information Technology	
F	Front-End Complex	CIM-OSA
FRB	Formal Reference Base	CIM-OSA
HF	Human Front-End Services	CIM-OSA
I	Information Complex	CIM-OSA
IEE	Integrated Enterprise Engineering environment	CIM-OSA
IEO	Intergrated Enterprise Operations environment	CIM-OSA
IIS	Integrating Infrastructure	CIM-OSA
ISA	Integrated Systems Architecture	
ISO	International Organization for Standardization	
LOTOS	Language of Temporal Ordering Specification	
MF	Machine Front-End Service	CIM-OSA
ODP	Open Distributed Processing	
OSI	Open Systems Interconnection	
PbLOTOS	Probabilistic LOTOS	
PDU	Protocol Data Unit	
PrLOTOS	Priority LOTOS	
PS	Protocol-Support Service	CIM-OSA
RM	Resource Management Service	CIM-OSA
SD	System-Wide Data	CIM-OSA
SDU	Service Data Unit	
SE	System-Wide Exchange	CIM-OSA
SP	Service-Provider	CIM-OSA
TLOTOS	Timed LOTOS	
TPN	Timed Petri-Net	
X	(used as a placeholder name for any IIS service)	CIM-OSA
XL	Extended LOTOS	
(X)L	(Extended) LOTOS	