THE DESIGN OF DISTRIBUTED SYSTEMS USING PARTIAL CONSTRAINTS

Ramesh Ahooja

A Thesis

in

The Department

of

Electrical Engineering

Presented in Partial Fulfillment of the Requirements
for the degree of Doctor of Philosophy at
Concordia University
Montreal, Quebec, Canada

June 1991

ISBN 0-315-80980-9

ABSTRACT

The Design Of Distributed Systems Using Partial Constraints

Ramesh Ahooja, Ph.D.
Concordia University, June 1991

Engineering the design and formal specification of 'open' Distributed Systems and Protocols is considered. The complexity of Distributed System behaviour makes design difficult, often resulting in intractable algebraic specifications. Verification of infinite behaviour using Bisimulation Equivalence has been shown to be undecidable. The *partial constraint methodology* of design and specification is elaborated in this thesis. The novel and original notion of *partial constraints* is used in the strategy of designing systems with distributed behaviour. Partial constraints result in partial specifications tractable to verification analysis. The design strategy relates actions of a distributed system in a communication environment to requirements and system architecture. A satisfaction relation between partial constraints and generic process algebraic behaviour expressions is defined. The design relation provides guidelines for system design and specification, while the satisfaction relation forms the basis of verification. The method proves to be powerful enough to capture complex procedures of important protocols such as the OSI Transport protocol in algebraic specifications. The tractability of verifying resulting specifications is illustrated by means of the OSI Transport and the Alternating Bit protocols.

DEDICATED TO MY PARENTS

BALWANT KAUR AND RAMSINGH

AND TO MY WIFE JOSEE AND OUR CHILDREN

Acknowledgements

Acknowledging the contribution of all the people who made the writing and completion of this thesis possible is an immensely gratifying task for myself. In the time it took to write this thesis, I received the support and encouragement of many people.

Without the help of Professor P. Ziogas-Chairman, E.E. Department, this thesis would not have seen the light of day. My heartfelt gratitude goes to him. Professor Ziogas is that rare academician, whose extremely sensitive and just attitude towards graduate students will always be remembered.

I thank Dean M.N. Swamy for his crucial support. His support enabled me to finish my program in a very successful manner.

I thank my supervisor Professor Dr. K. Thulasiraman. Professor Thulasiraman encouraged me throughout the writing of this thesis. Support from a person of such immense technical competence and ability were a major motivation for me to complete this thesis. I thank Professor R.P. Zeletin, for generously allowing me to work in execellent facilities for research activities in open communication systems.

I thank Professor L. Logrippo-University of Ottawa, for being a friend and mentor (even though he is reluctant accept the mentor role). Dr. Logrippo gave me the idea of writing a thesis on system design.

I thank Dr. B. Sarikaya, for supervising me in the early part of the program.

I thank Dr. Anindya Das, for spending a lot of his valuable time, reading and commenting this thesis. His suggestions were extremely insightful, sometimes leading to discussions which greatly improved the quality of this thesis.

I also thank S. Palacherla and R. Prasad for providing me with logistical support during my frequent trips to Montreal for this thesis.

And finally, I thank, Professor C. Vissers, G. Scollo, E. Brinksma, E. Najm, T. Bolognesi, and all the colleagues and friends associated with LOTOS in Europe and north America whose work in the area of formal specifications made this thesis significant.

Of course this acknowledgement would not be complete if I did not thank my comrade in life, my wife-Josee Biard, whose constant help, and backing through thick and thin with unceasing care of our children, made it all possible.

Table of Contents

LIST OF FIGURES AND TABLES

FIGURE

LIST OF SYMBOLS

$\Box$	Modal operator 'henceforth'
$\Diamond$	Modal operator 'eventually'
$--\alpha-->$	Transition of a process from with observation of an action α.
$\alpha,\beta,\gamma...$	Atomic actions of an Environment
ε	Belongs to
$\{A_i\}$	A Heterogenous Algebra
Ω	A collection of Operations of a heterogenous algebra
ω	An operation of a heterogenous algebra
τ	The silent unobservable action of a CCS process
i	The silent unobservable action of a LOTOS process
+	The CCS,ACP Choice operator
[]	The CSP,LOTOS Choice operator
|	Parallel Composition of CCS,ACP
||	Parallel Composition of LOTOS,ACP
|||	The LOTOS interleaving operator
δ	Successful termination action of
NIL	The null action
E_X	Guarded Recursion Equations
	LOTOS, or encapsulation operator of ACP

E	A CCS behaviour expression
$\sim_0$	"0-equivalence" of Processes
$\sim_k$	"K-equivalence" of Processes
$\cap$	Set Intersection
$\approx_c$	Congruence
or	Non-deterministic 'or'
and	Communication 'and'
|=	Derives, satisfaction relation
==>	implies
$\equiv$	equivalent to
$\forall$	For all
$\exists$	There exists
<	Less Than
$\leq$	Less Than or Equal To
wff	well-formed formulae
ν	The Non-deterministic relation between actions
μ	The Communication relation between actions
χ	The Concurrency relation between actions

LIST OF ACRONYMS

LTS	Labelled Transition System
ISO	International Standards Organisation
OSI	Open Systems Interconnection
BRM	Basic Reference Model
PE1	Peer Protocol Entity 1
PE2	Peer Protocol Entity 2
FDT	Formal Description Technique
LOTOS	Logic Of Temporally Ordered Systems
ESTELLE	Extended Finite State Machine Specification Language
SDL	System Description Language
CCS	Calulus of Communicating Systems
CSP	Communicating Sequential Processes
ACP	Algebra of Communicating Processes
OBJ	Data Type Specification Language
ACTONE	Data Type Specification Language
FTS	Finite Transition System
DT	A CCS Derivation Tree
ASP	Abstract Service Primitive
SAP	Service Access Point
FSM	Finite State Machine
ADT	Abstract Data Type
ABP	Alternating Bit Protocol
Pc	A set of Partial Constraints
pc	A partial Constraint

"Wipe your glosses with what you know"

- James Joyce

Chapter 1

INTRODUCTION

1.1 Nature Of The Thesis

This thesis is on the design and specification of Open Distributed Systems. Computer Systems with parts distributed over geographically separate computers which use underlying services provided by physical communication networks, to communicate and co-operate, are called distributed systems. The International Organisation for Standardisation (ISO) considers a system to be 'open', if it adheres to recommendations contained in ISO-Standards documents for Open Systems Interconnections (OSI). OSI standards describe behaviour mechanisms which enable real computer systems to communicate and interwork in a network of heterogeneous computers to achieve a common (distributed) task. To support the development of open distributed systems ISO has provided standardised architecture in the form of the seven layered Basic Reference Model (BRM) [41]. Architectural details of the BRM are discussed in the sequel.

Two decades of maturing coupled with a profound integration of two distinct but complimentary technologies has resulted in an explosion of communication services. The breathless pace with which the modern computer has come of age in computational power is equalled only by the unimaginable capacity and variety of communication services offered by the other, the telecommunication network. The new integrated technology

called 'telematics' in turn has led to widespread activity in the development and use of distributed systems.

In telematics, underlying telecommunication networks provide communication services to distributed systems. Distributed systems development implies the combination of computer and network services. The issue of design and implementation of such systems gives rise to simultaneous orthogonal challenges: that of developing a powerful tool, indispensable to information systems, and yet complex to design and analyse.

Users of a distributed system make use of an information system with a distributed implementation. On the one hand, the system appears to be virtually homogenous, though, in practice, it is made up of heterogenous parts distributed over a wide variety of real systems with different capabilities and resources. On the other hand, the parallel processing which inevitably occurs on the geographically separated real systems, enforces complex concurrency constraints on system design and analyses.

Distributed systems have found popular application in the areas of database systems, parallel processing systems and communication systems. Communication systems exchange information between remote systems, in the form of voice, video and data.

1.2 Motivation For The Thesis

Distributed systems are designed to provide services to users. The characteristics of the service provided is determinable by composing the services provided by the

different component sub-systems. The objective of obtaining the required service makes distributed systems difficult to design, and complex to specify.

It is widely accepted that specifications which are rigorous and well-defined, compared with informal ones, are easier to analyse for validating system services.

Distributed system services are validated by checking for correct designs and complete specification relative to user requirements. User requirements are usually obtained from requirement descriptions. A demonstration, that system design features and system behaviour specification is of correct intention with respect to user requirements, validates the system. Proving that system operation, as derived from the specification is correct, verifies the system. Verification results in determining which logical properties are possessed by the system. Logical properties of systems can be used as the basis of validation.

Rigorous and well-defined system specifications are obtained by using specification techniques based on formal models of describing distributed system behaviour. The Finite State Machine (FSM) model [11] and the Algebraic model [75],[13],[45],[70], are the two principle models used in formal description techniques of distributed systems.

Validation of formal model based system specifications is also called 'model checking'. The tractability of a formal system specification is of concern in model checking. Strategies for reducing the number of states in reachability analysis of FSM based description techniques have resulted in tractable verification analysis of systems. However, reachability analysis becomes intractable in the case of systems with large numbers of states.

For process algebraic model based specifications, the notion of observational equivalence of communicating processes introduced in [54] was anticipated to be the most useful for verification. This idea forms the basis of deciding the degree to which two system specifications are similar. However, deciding observational equivalence of algebraic specifications describing complete behaviours of distributed systems, appears to be difficult [44]. Attempts to verify algebraic specifications of simplified or partial behaviours has met with some degree of success. In [12] and [59] for example, the use of observational equivalence for verification purposes, proceeds with the generation of FSMs of the given algebraic specification. Such an approach to validation of algebraic behaviour specification ultimately relies on verification analyses of FSMs.

A method of validation of algebraic specifications of processes using Modal Logic [39] has been described by R. Milner in [55]. In [55], he has also shown that Modal Logic assertions satisfied by observationally equivalent behaviour specifications are equivalent. The results reported in [55] form the main inspiration for the writing of this thesis.

1.3 The Scope Of The Thesis

This thesis is concerned with issues emerging in the development trajectory of Distributed Systems software. Design and specification forms the first phase of the development trajectory of distributed systems software. The second phase usually includes validation and verification of the specified system. The third and last phase is implementation of a real system, which has to be validated by testing. These last two phases are intended to gain a measure of confidence in the final output of the development trajectory - usable software.

Each phase of the development trajectory has a corresponding identifiable phase objective. Techniques found to be simple and useful for the objectives of design, specification and verification of distributed systems, are developed in the following chapters.

The distributed system design and specification techniques developed herein are elaborated in the framework of the OSI BRM. The BRM describes a seven layered architecture with each layer providing a communication service to the layer above it. The model is detailed in Section 1.5.3 of this chapter.

In most Engineering disciplines system design and specification proceeds on the basis of constraints on system behaviour. Constraints are obtained from user requirements. This thesis develops the *Partial Constraint Methodology* for design and validation of Distributed Systems. Formalisation of the notion of *partial constraints* presented in Chapter 4 is a new and an original contribution of this thesis. A formal definition of partial constraints on the events associated with a distributed system in a communication environment, serves as a tool to specify system behaviour. The design of a distributed system based on the Partial Constraint Methodology, results in specifications which satisfy the objectives of design. If validation is the objective of system design, then Partial Constraints aid in the statement of a set of design criteria for validation. A validation method based on partial constraints is developed in Chapter 5. If verification is objective of system design, then Partial Constraints aid in the verification of the logical properties of the system. A verification method is developed in Chapter 6 of this thesis.

This thesis does not concern itself with issues of distributed system performance, or issues involving the stochastic analysis of either the systems or the efficiency of protocol implementations and the physical networks they use. Nor is it concerned with issues related to the implementation of distributed system software in specific, real computer environments.

1.4 Contributions Of The Thesis

The main contribution of this thesis is in the area of Distributed Systems and Communication Protocols. The contribution takes the form of a design methodology for distributed systems called the Partial Constraint Methodology. The Partial Constraint Methodology is guided by the view that system design must be constrained by the design objectives of validation, verification and testing.

Specifically, this thesis makes a contribution to the area of Distributed System and Communication Protocol design by:

1. Relating design objectives to design methodology.
2. Formally defining the notion of partial constraints on distributed system behaviour. Partial constraints are used as design and specification tools.
3. The development of a methodology for obtaining process algebraic service specifications from protocol specifications.
4. The development of a methodology which uses formal partial constraints to validate specifications in the design-to-specification phase of the development trajectory.

5. The development of a methodology which uses formal partial constraints to verify specifications in the design-to-specification phase of the development trajectory.

1.5 The Development Of Open Distributed Systems

Open distributed system development begins with the design and specification phase. Validation and verification concerns are usually treated in the next phase, followed by implementation and then testing of the 'real' (physical) system.

1.5.1 System Design Development

System design consists of mapping requirement intentions and notions onto formal expressions of system behaviour. System behaviour is expressed in a specification language which is based on an underlying model of behaviour. Structuring specifications of large systems improves the tractability of analysis. Experience [63] shows that stepwise refinement of the specification structure by the gradual addition of more detail relevant to a target implementation environment helps in implementing large systems correctly.

1.5.2 Formality Of The Model For Design

All computer programming languages are based on formal models. The formality of a program language model is dependent on the environment in which it is implemented. This implies that all software systems are specified formally, even though, the level of abstractness of the formalism is commensurate to its implementation environment. Proper concern for the formal model used in every phase of the development trajectory improves

the quality of the output obtained from that phase. The level of abstractness of the formal model, and therefore its degree of independence from the implementation environment, in each phase of the trajectory can be determined by the level of abstractness of the required output - a formal "specification" of the behaviour of the system.

In the design and specification phase, the formal model used for specification must be as abstract, or as independent as possible from the physical and system application environments in which the system is targeted to be implemented. This is due to the heterogenous nature of the physical systems and application environments which communicate and interwork via open systems interconnections. An abstract model determines the design, and forms the basis of analysis of system behaviour specifications, and their use in the form of different implementations in varying environments. Independence of such a model is obtained by abstracting away from (suppressing irrelevant information details) dependencies considered to be irrelevant to the design and specification task.

Thus, for example, physical dependencies include specific computer architectures and their machine languages, and at the next higher level of abstraction, even computer programming languages. While system application specific dependencies may include the essential characteristics of real-time systems, data base systems, knowledge based systems and even that of distributed systems.

1.5.3 Criteria For Model Selection

The main criteria for the selection of a particular formal model are, that it be compatible with design objectives and provide a set of convenient design tools. The formal

model should not only permit abstraction from dependencies, considered to be unnecessary in the design phase, but also support system description with its ability to precisely model features considered to be essential to the specification of the system.

For distributed system design, the abstract formal models which are of interest to us are those which are independent of physical dependencies, and all inessential application specific dependencies. Therefore, the abstract model used for design must express the essential features of abstract behaviour.

System architecture is the feature which dominates all other features which are required to be formally modelled. The essential characteristic of distributed system architecture is distribution of system parts. Communication and cooperation between distributed system parts occurs with the participation of an underlying service - a non-distributed entity. The combination of system components and underlying services provide the required service to users of the system.

For example, the OSI BRM architecture forms a framework for open distributed systems design. The layered architecture of the reference model assumes an underlying service provided by each layer to protocol entities of the layer above. The combination of protocol entities and the underlying service provides the required service to users in the next higher layer. Requirement descriptions of layer service and protocol entities are described in the OSI standards documents in terms of behaviour mechanisms (see for example: [41]).

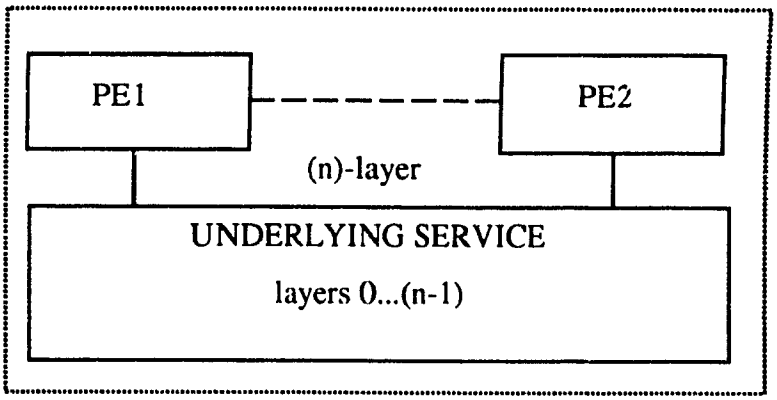


Figure 1.1. Layered architecture of the OSI reference model

Figure 1.1 shows the OSI reference model seven layered architecture, in which the service consisting of the combined services of layers 0 to (n-1) is shown as the underlying (n-1)-service. PE1 and PE2 are protocol entities which communicate and cooperate to form the (n)-layer protocol. The line connecting PE1 and PE2 represents the logical link constituting the (n)-layer protocol. Communication and cooperation behaviour between PE1 and PE2 logically takes place via this protocol link and physically via the underlying service. The combined behaviour of the protocol entities and the underlying service results in the (n)-service, shown as the outer dotted-line rectangle of Figure 1.1.

The seven layers of the reference model are well known as the physical, data link, network, transport, session, presentation and application layer respectively.

1.6 Essential Features Of Distributed System Behaviour

In a layered environment such as OSI, design activity begins with requirements and ends with a formal definition of system behaviour. The formal definition of behaviour, called a formal specification of the system is based on a selected formal behaviour model.

A reference architecture for distributed systems such as the one described in Section 1.5.3, and the need for formal specification of system behaviour has aided in the identification of features [55],[13] considered to be essential in a model for distributed systems design. These features are well known as the concepts of:

1. Non-determinism

A distributed system exhibits non-deterministic behaviour. This means that, to an observer, the system appears to choose between alternative behaviours autonomously; The choice being made without any apparent (to the observer) causal relation between a chosen alternative, or between the system and its environment.

2. Concurrency (sequentiality)

Concurrent behaviour is the reduction of non-deterministic alternatives to a simpler non-deterministic choice between sequential behaviours. The system chooses between either of two alternatives concurrently. The chosen alternative precedes the other - followed in sequence by the other. That means that the possibility of only one of the alternatives occurring without being followed by the other, is removed from the possible non-deterministic choice of behaviours. Sequential behaviour is a further reduction of concurrency, to that of a sequential order. The presence of a causal relationship determines the ordering of behaviour into an appropriate sequence.

3. Communication (Synchronisation)

When two concurrent behaviours have common elements they can communicate (or synchronise) on the basis of sharing or exchanging the common element. The common element could be an interface or a data message or both. Since different parts of a distributed system must communicate and cooperate, a model of synchronisation between behaviours whereby messages and data values can be exchanged is essential to formalise.

4. Composability

The behaviour model should allow the composition of behaviours of different parts of the system in a consistent manner. That is, the composed behaviour should consistently represent the combined behaviour of the individual parts without contradicting the original parts.

5. Distributability

Distributability is a structuring property, whereby a monolithic system is decomposed. That means it should be possible to structure behaviour into independent and dependent parts such that complexity becomes manageable, with dependencies clearly represented by the structure. Independent parts can then be separated and developed by autonomous agencies.

1.7 Design Objectives

A behaviour model selected for design and specification must have some additional features as well. One of the aims of this thesis is to show how design objectives play an important role in the design and specification process of each phase of the development trajectory. The lack of a clear identification of design objectives in any phase of the development trajectory adds to the complexity of system development. A proper concern and identification of clear design objectives in every development phase, results in well-defined and correct formal specifications - in a style suitable for the satisfaction of such objectives.

The proper choice of a behaviour model and associated tools, such as a specification language, considerably ease difficulties of system design.

The OSI BRM architecture serves as an exemplary basis for the design of open distributed systems. In the context of such an architecture, the following design objectives are important:

1. Service Specification

Service specifications [69], [79], [71], [80] describe the services obtained by the users of the system. Users in a certain layer, use the service provided by the system consisting of a combination of services provided by the underlying layers. A convenient architecture of the system viewed as a service provider, is that of queues transmitting messages between users (see for example [42]).

2. Protocol Specification

Specification of the protocol [11], [74], [71] for message exchange in a certain layer, which provides services to users in the layer above. To account for the distributed structure of the protocol, the behaviour definition of the system consists of the behaviour definitions of the protocol entities of that layer. The protocol is defined in such a way that it uses the services of underlying layers to provide the service required by users in the next higher layer.

3. Combined Specification Of The Protocol And Underlying Service

Combination of the behaviour definitions of the protocol entities and the underlying layer service, define the services provided to users in the next higher layer.

4. Verification

The concept of verification of distributed systems evolved from verification of computer programs [27], [36]. Verification implies correctness of system behaviour. Proving that logical properties of the service required from the system, are indeed properties of the system's behaviour as it is specified - is verification. Verifiable logical properties of behaviour can be reduced to generic properties of distributed systems called Safety and Liveness properties [61], [31], [49]. Safety properties have the form "bad things will not happen" and Liveness properties have the form "only good things will happen". A safety property asserts what the system is allowed to do according to its

specification, or dually, what it may not do. Liveness properties describe what the system must do (eventually) according to its behaviour specification [54], [48].

5. Validation

Validation is interpreted in this thesis, to imply a demonstration that the behaviour specification resulting from the combination of the specification of the components of the distributed system provides correct service to users. Validation, therefore implies both, syntactic and semantic consistency, between specification of components. Techniques for demonstrating validity include simulation and testing methodologies. For protocols, consistency between protocol specification and underlying service specification (third objective above) implies validation of the system specification. If additionally, the system specification is also consistent with the service specification, then the system is considered to be valid against intended behaviour. That is, the combined behaviour is the one which was intended, in order to provide the required service. This amounts to validation of the protocol behaviour against correct intentions.

6. Testing

Testing the physical implementation of a distributed system by experimentation under laboratory conditions also validates system behaviour against requirements. Test experiments on implementations usually have the form of stimulus-response sequences [28],[35]. Stimulus-response sequences are obtained either from a 'reference' behaviour specification, from which an implementation is assumed to be derived, or from requirements descriptions. Testing a system in order to validate its behaviour against

standardised requirements, such as those of ISO-Standards is generally referred to as conformance testing [14].

7. Testing Protocol Performance

Testing physical implementations using stochastic models of behaviour with the objective of improving the quality of service is referred to as performance testing.

The relative importance of these design objectives in the design of a distributed system provides valuable hints for a judicious selection of design tools and methods.

Thus, the first design objective stated above implies that the specification of system behaviour resulting from the design process must be in a formal language (or description technique) which is able to express those concepts which are important to the environment of user requirement descriptions. The second and third objectives stress the need for a model which allows composability of simple behaviour structures into more complex ones. Composability also implies its converse, namely - abstraction. Abstraction mechanisms reverse in some aspects the effects of composition. In some contexts, they reduce the information about the system by those details which are considered to be irrelevant to furthering the design. The reduction effects both the structure and semantics of the specification. Design objectives number four and five imply that it should be possible to have the resulting specification, and indeed specifications intermediate to the design process, in a language and form which allow us to reason about the system.

The sixth design objective is left for future elaboration. The seventh design objective is considered to be outside the scope of this thesis.

1.8 Contents Of The Thesis

The rest of chapter 1 is devoted to outlining the contents of tne following chapters. *Chapter 2* introduces the necessity of selecting a behaviour model for generating formal specifications. Such a model must also, accurately capture the essential notions of distributed behaviour. *Chapter 3* briefly surveys existing behaviour models and interprets them uniformly using the semantics of heterogenous algebras.

The development of the methodology of distributed system design and formal specification is begun by summarizing the concepts of system services and their relation to architectures in Chapter 4. The relationshio between services, design and constraints is also explored in this chapter. The chapter ends with a formal definition of partial constraints followed by illustrative examples of the design of simple systems. *Chapter 5* develops the relationship between partial constraints and formal specifications in process algebras. The methodology of partial constraints for system design and behaviour specification is shown to be intimately linked to validation of formal specifications. *Chapter 6* develops the validation and verification methodologies - illustrated with real benchmark protocols - the Alternating Bit Protocol (ABP) and the OSI Transport Protocol. *Chapter 7* provides the conclusions and possible directions of future work.

" 'Fetch me from there a fruit of the Nyodgradha tree'
'Here is one, Sir'
'Break it'
'It is broken'
'What do you see there?'
'These seeds, almost infinitesimal'
'Break one of them'
'It is broken'
'what do you see there?'
'Not anything, Sir' "
- From the Chandogya Upanishad

Chapter 2

MODELS FOR BEHAVIOUR SPECIFICATIONS

In this chapter formal models for distributed systems behaviour specifications are discussed. Characteristics which are important for a behaviour model intended for use in design and specification, are identified. The domains of model application are defined. Simple examples are used to illustrate the notions of validation and verification.

2.1 Formal models

Formal models for behaviour specification relate semantical domains of notions and intentions to formal specifications. The theory of computation underlying the model defines the semantic concepts required to express essential ideas of system design. Computation theories, such as Automata theory [38], Net theory [66], Mathematical Logic and Function Theory [16],[17],[51], have been used to model the concurrent behaviour of distributed systems.

The complexity of system behaviour has led to the study of different models and techniques of design and specification. Transition system models [2] have been widely

used to design and specify concurrent behaviour. Examples of description techniques based on a transition system model are Petri-nets, FSMs and the OSI Formal Description Techniques (FDTs) [23] - LOTOS [25], ESTELLE [24] and SDL [26]. Formal specification languages are called FDTs within ISO.

Logical models of specification, such as the Temporal Logic [66] and Predicate Logic [52] are based on the mathematical theory of Modal logic [39] and Predicate Calculus [51] respectively. Logical models are mainly used to verify those properties which a system is required to satisfy. Examples of the use of Temporal logic models and Predicate Calculus models are discussed in [31] and [17] respectively.

Among the models based on transition systems, Petri-net models model concurrency adequately, but net descriptions of system behaviour invariably have to be reduced to their state machine equivalents for verification analysis. Thus, the inadequacies of state machine models becomes unavoidable for verification purposes.

The development of a Calculus of Communicating Systems (CCS) [54], a process algebraic model, marked the beginning of the study of a paradigm for concurrent computation in order to "treat concurrency on its own terms" (Milner in [55]). CSP [35] another process algebraic model for concurrent systems highlighted the importance and power of using the method of defining systems of equations of algebraic processes in system modelling. Process algebras are discussed in Section 3.2 of Chapter 3.

The application of algebraic theory to the specification of computational systems, began with the development of Abstract Data Type Specification (ADT) techniques [29]. This seminal work applied the wide experience and knowledge of algebraic methods of

mathematics, "with full force" [50], to provide a rigorous semantical basis, via Initial Algebra semantics, to the specification of data types. Algebraic Type specifications have been used in early computational systems such as OBJ and Alphard [83]. Initial Algebras are explained in [29]

Computational systems semantically, are mathematical objects in algebraic model based system specifications. Type parameterisation [77] and value passing [34] between type specifications enhances the power of the algebraic type model of specification. These powerful specification techniques are complimentary to the constructive type specification methods found to be useful in the design, validation and verification of algebraically specified distributed systems. Examples of the use of such methods are reported in [30].

The most significant use of a type specification model based on initial algebra semantics is the FDT ACTONE [22]. ACTONE is a specification language which provides a complete set of notational constructs for the specification of typed objects. Parameterisation and type combination is also possible in ACTONE, making it a very powerful and flexible specification language. An example specification in ACTONE is given in the latter part of this section.

More recently, an Algebra of Communicating Processes (ACP) [7] has contributed towards an initial algebra semantics for concurrency in process algebraic models, in an attempt to parallel developments in abstract data type theory.

The FDT LOTOS, is an example of the use of an algebraic model based definition of a specification language intended for open distributed systems. It consists of two complimentary components, both are based on algebraic models of system specification.

One component models system behaviour in terms of a process algebra. The process algebraic component of LOTOS is based on CCS and CSP. The other component is ACTONE, used for communication of data values when processes synchronise. The behaviour part of LOTOS provides axioms and inference rules for semantical interpretations of the dynamic behaviour of processes.

In the algebraic framework, a formal model is a set of mathematical expressions and rules for the creation of abstract objects. Abstract objects can be manipulated according to such rules, given in the form of axioms or equations. For distributed systems, an algebraic model is a set of operations and expressions with rules and/or axioms defining the results of manipulating expressions of system behaviour. Systematic assumptions and properties of a model derivable from methodologies are called 'theories' about the model. For example, the assumptions of a given model of benaviour specification may support the development of a technique (as we shall see later), to verify logical properties from expressions of system behaviour.

In this thesis, the framework used for the description of models of distributed system behaviour is that of heterogenous algebras [50]. A heterogenous algebra is an abstract operational model of a system. It views the system as an algebra defined with operations on many different carriers. Carriers are characteristic sets of values of a certain type of mathematical object. A precise definition of a heterogenous algebra is given in the sequel.

In what follows, some general requirements expected of a formal behaviour model for distributed systems are discussed. This is followed by a toy example to give a "feel" for some of the abstract notions related to formal models.

2.2 General Requirements For Formal Models

The objectives of system design as well as the essential features of distributed system behaviour were identified in the previous chapter. A consequence of such an identification is a characterisation of the general requirements on behaviour models used for the specification of system behaviour.

General Requirements:

1. Well-definedness

A formal model should be well-defined. That means, behaviour expressions based on the model should result in unique and unambiguous interpretations of system behaviour.

2. Comprehensibility

A formal model should have a few simple but precise conceptual tools for comprehensibility. The conceptual tools must be easy to use and simple to understand. The tools should also, precisely capture notions and intentions for which they are designed.

3. Generality

A formal model should be powerful enough to capture a broad range of concepts, but also be flexible, for representation of a wide variety of intentions and notions for generality.

4. Orthogonality Of Concepts

A formal model should permit orthogonal concepts for extensibility and reducibility of semantic information in a specification. A specification based on such a model should be extendible by the consistent addition of semantic information. Abstraction mechanisms should allow the reduction of semantic information from the specification without leaving the resulting specification with conflicting elements.

5. Structuredness

The model should include structuring concepts for dealing with complexity [20], to aid in analytical activities such as verification, validation and testing.

2.3 Application Specific Requirements

For distributed system design a formal model should incorporate the following concepts:

1. Expressiveness

The model should have the expressive power to express notions of non-determinism, concurrency (sequentiality), and communication between behaviour parts.

2. Composability

A model should observe the principle of composability for consistency over all parts of a distributed system in order to permit validation of specifications.

3. Distributability (Vertical Decomposition)

The requirement of distributability on a formal model, though similar to that of structuredness (horizontal decomposition) and implied by that of composability is distinctly identified here to stress the importance of being able to view a distributed system, virtually, as a monolithic system from the point of view of the service obtained from it. And also, the model should provide methods which would hint at proper internal structuring of a virtual monolithic specification. The internal identifiable structures should aid in the eventual independant and distributed implementation of independent components.

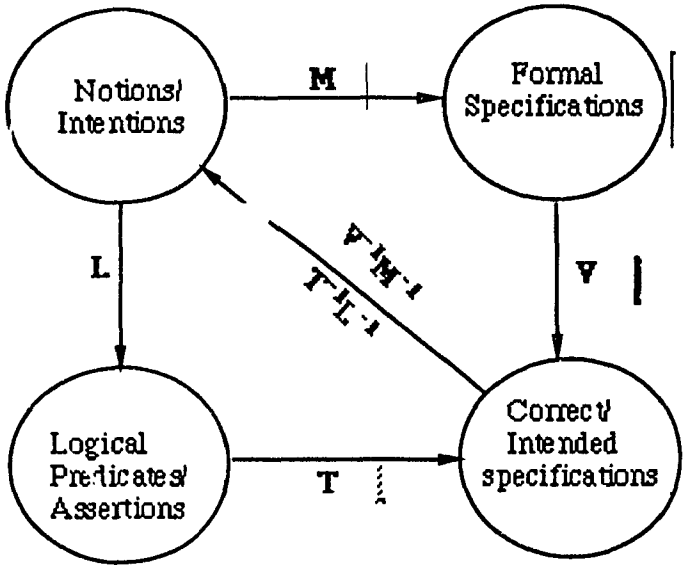
4. Completeness

A formal model should be complete. Completeness ensures that no essential concepts of distributed systems have been excluded from the model. An associated

property is exactness, which implies that the model allows for mechanisms to avoid over or under specification.

2.4 Domains Of Formal Models

The domains of formal models of behaviour are represented by a mapping diagram in Figure 2.1. In Figure 2.1, system specifications are semantical objects built with the conceptual tools provided by a model for behaviour description.



M = Semantic Model for Specifications

V = Validation or Verification

L = Logical functions

T = Truth Functions

Figure 2.1. Domains of formal models of behaviour

In Figure 2.1, M, V, L and T are relations between pairs of values from the domains of notions/intentions (informal), formal specifications, logical predicates or assertions and correct or intended specifications.

A behaviour model M is a relation which is able to associate notions and intentions of design to formal specifications of system behaviour. An example of M is the process algebraic model of CCS.

V is a verification relation on correctness properties of system behaviour. A verification technique V proves a satisfaction relation between properties of system behaviour as derived from its formal specification and required logical properties derived from descriptions of intentions. That is, V can take a required behaviour property (or an intention) and a specification pair, and return a boolean value of true or false, depending on whether the formal specification satisfies the required logical property or not. Verification techniques depend on both, the behaviour model and the technique of relating specified behaviour to logical assertions.

Usually the generic functional properties - Safety and Liveness are verified [31]. For example, given the behaviour specification of the (n)-service in Figure 1.1, correct delivery of data sent by one distributed user to another is a verifiable safety property. And eventual delivery of data sent by a user is a verifiable liveness property of the provided service.

If the formal specification has an implementation, then V is a validation relation between the implementation and correct intentions of desired service. A system implementation can be in the form of a formal specification. The validation function V,

demonstrates, by simulation for example, whether the behaviour as represented by the specification is correct as intended. That is V can take an implementation specification and intended specification pair, and return a a boolean (true or false) depending on whether the implementation specification is syntactically and semantically consistent relative to the intended specification. For example, showing that the (n)-service, obtained by the consistent composition of the behaviour of system parts (the protocol entities and the underlying service in Figure 1.1) is syntactically and semantically consistent with an independent description of the intended (n)-service constitutes validation.

L is a logical model of behaviour which can correctly relate notions and intentions of systems design to logical predicates and assertions on system behaviour.

An example of a logical model is the temporal logic description of the Alternating Bit protocol in [31]. In such a model, system behaviour can be described in terms of temporal operators. Temporal operators correspond to predicates characterising future behaviour of the system from a given starting point. The predicates describe system behaviour in terms of the operators: henceforth (denoted as '□') and eventually (denoted as '◊'). The semantics of 'henceforth' is that a given property will henceforth (from a given point in the computation) always be true. And that of 'eventually', is that the given property will become true sometime (including the present time) in the future.

T represents a system of truth functions associating logical and predicative descriptions of behaviour to intended specifications. The truth functions are able to take a logical assertion (or predicate) on system behaviour and intended specification pair, and return a value of true or false according as the assertion satisfies the intended specification or not.

For example, given a logical assertion in terms of first order predicate logic, the Lambda Abstraction Functions [56] can be defined to return a value of true or false for the given assertion.

The composed relations $V^{-1}M^{-1}$ and $T^{-1}L^{-1}$, if they exist, relate correct and intended specifications to the original notions and intentions of design.

The existence of undecidable problems in computational theory often lead to specifications in any behaviour model which are impossible to verify or validate. It becomes therefore necessary to make suitable assumptions in the model to allow only the class of decidable problem domains for specifications. This can be done in the axiomatic approach for example by making suitable assumptions to restrict the specifications generated by the model to the class of decidable computations.

2.5 A Conceptual View Of Validation And Verification

From the above it becomes clear that a model provides conceptual tools usable for formal behaviour specifications. A formal specification language is one such conceptual tool. The formal specification language determines the form of the specifications it generates, but it is the behaviour model which substantially influences the methods used for the validation and verification analyses of the specification.

In order to illustrate and concretise some of the concepts discussed above in a non-technical manner, a 'toy' example is given below.

Example 2.1

To illustrate that the form of a system specification is determined by the conceptual tools provided by a formal model, a "toy" example is given below:

The first example consists of a blank paper on which curved lines can be drawn possibly with the help of other tools such as a pencil and compass. Then Figure 2.2 shows the formal specification of a circle based on the given model.

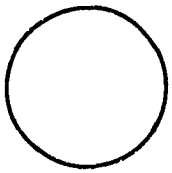


Figure 2.2. Formal specification of a circle

Assume that the tools of a second model consist of certain graphic schemata shown in Figure 2.3.18

Figure 2.3. Conceptual tools of a description model

Then the formal specification of a circle in this model is given in Figure 2.4.

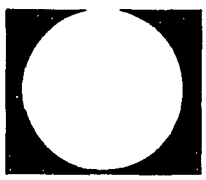


Figure 2.4. Formal specification of a circle with graphic schemata

Any circle has the property that, it has the same curvature throughout its circumference. This property can be verified by superimposing the circle of Figure 2.2 on Figure 2.4.

If the circle was specified as two distributed halves, with each half consisting of one piece of the schemata of Figure 2.4, then composition of the two halves and a demonstration that the resultant specification corresponds to the circle either of Figure 2.2 or that of Figure 2.4 would validate the specification against correct intention.

In this case, correct intention is the requirement of obtaining the appropriate semantical object - a circle.

[end of example 2.1]

The circle specification provides a useful illustration of how proper description tools of a given model can support design based on verification and validation objectives.

For example, the conceptual tools for specifications shown in Figure 2.3 can also be considered to be 'constraints' on the semantical objects which are the target of specification. The specification tools of Figure 2.3 "constrain" the specification of a circle given in Figure 2.4.

The idea of using constraint descriptions as a guide to the design and specification of distributed system has been widely used [79], [60], [81] albeit without a formal definition of the notion of constraints. Constraint-Oriented specifications form the basis of showing relationships between styles of specifications and proving behavioural equivalences between processes for verification [81].

In chapter 4, a model of *partial constraints* is introduced as a conceptual tool to aid in the design and specification of distributed systems. The notion of partial constraints is new and original. Logical relationships between events of a system are defined in a communication environment. Definition of event relationships on system behaviour formalise the notion of partial constraints .

The partial constraint method of specification maps partial constraints onto process algebraic behaviour expression constructs, such that specifications induced by the process algebra systematically satisfy partial constraints defined during design.

In the next chapter behaviour models for distributed systems are summarised in order to lay the basis for partial constraints defined in the next chapter.

"A different name has a different meaning, a different meaning has a different name, for - for that one person. But the same name has a different meaning for different people, and different and opposite names may have the same meaning for different people".

- R. H. Blyth

Chapter 3

DISTRIBUTED SYSTEM BEHAVIOUR

In this chapter, behaviour models of distributed systems are discussed. Important features of distributed systems, viz: non-determinism, concurrency, sequentiality and communication are examined in the light of descriptions in state machine models, data type models and process algebraic models. The underlying similarities in the description parameters of the specification models are presented in the framework of Heterogenous algebras.

3.1 Behaviour Models Of Distributed Systems

From the point of view of logical behaviour formalisms, distributed systems are no different from any other kind of system (cf. [49]). However, algebraic analyses of distributed systems often involves transformations made on non-deterministic specifications. For example, non-deterministic choices in a specification may be reduced to concurrency in an implementation or even sequential behaviour.

Transformations on different specifications based on different models, lead to different interpretations of the notion of non-determinism. The differences are manifested in the parameters of the behaviour model.

Different models of behaviour use different combinations of notions for actions, processes and states (parameters of system descriptions). Indeed, different categories of models can be identified according to the description parameters used.

Thus, calling the mathematical abstraction of a system or a component of the system, a process P, the following notion of Labelled Transition Systems (LTS) for system behaviour can be used:

Definition 3.1

A Labelled Transition System (LTS) is a quadruple <*S*,L,T,s0>, where

S: is a countable set of states of P;

s0 ε *S*: is the initial state;

L: is the set of action Labels α, such that α ε L and

T ≤ *S* x L x *S* is a relation called the transition relation.

[end of Definition 3.1]

The notation s --α-->s' with α ε L is often used whenever (s,α,s') ε T, to describe the transition of P from state s to s' with the observation of the action α. If *S* is finite, the system is called a Labelled Finite Transition System (LFTS).

A different style of describing an LTS is one in which the action α is replaced by an input and an output action. This is often the case in descriptions using system states as the main parameter of description.

A labelled transition system can be described in terms of a heterogeneous algebra. This means that a heterogenous algebra can simulate an LTS specification. A heterogenous algebra is defined as:

Definition 3.2

A heterogenous algebra $[\{A_i\}_{i \in I} ; \Omega]$ consists of a family of different 'types' A_i of elements, together with a collection Ω of operations defined on these types in the following way: Associated with each n-ary operation $\omega \in \Omega$ ($n = n(\omega)$ is a (n+1)-tuple $(i_1,...,i_{n+1}) \in I^{n+1}$; ω is then a mapping (a heterogeneous operation):

$$\omega: A_{i_1} \times ... \times A_{i_n} \text{ ----> } A_{i_{n+1}},$$

thus the kth operand of ω is taken from A_{i_k} (k=1,...,n) and the value ω lies in $A_{i_{n+1}}$.

[end of Definition 3.2]

The definition characterises the set $A_{i_{n+1}}$ of data values called a sort created by the mapping ω. The notation '---->' can be read as 'maps to'

A heterogenous algebra simulates an LTS with the algebra defined by the tuple <*S*,L,T,s0>. *S* and L are carriers and T and s0 are operations:

s0: -----> *S* is a constant element of *S*

T: *S* x L x *S*----> *S* is the transition function T(s,α,s') that maps the system state s to s' with an action α.

System behaviour can be represented by sequences of transitions, such as:

$$P(s0) \text{ --}\alpha_1\text{--> } P(s1) \text{ --}\alpha_2\text{--}P(s2) \text{ --}\alpha_3\text{-->...}$$

where the α_i are atomic actions. The process P is parameterised by the state variable s_i corresponding to the set of states of P. The state s_i contains all information necessary to determine the future behaviour of the system.

The OSI Transport protocol specification in [41] is an example of an informal description using states and state transitions. The transport protocol is implemented in layer 4 of the OSI reference model architecture. The protocol can establish and release connections for the end-to-end transfer of data between remote user entities of the session layer above. Connection establishment and release as well as data transfer is accomplished by exchanging service primitives with session users and the underlying network service. End to end indications of connection establishment and release requests are given by protocol entities. The reception of data is acknowledged.

A heterogenous algebra which describes the state transitions of a transport protocol entity simulates an LTS description using input/output actions with state transitions. That is, the definition of the transition function has four arguments instead of three. The algebra is defined by the tuple [S,I,O,T] with three carriers and one operation:

S is the set of states,

I is the set of inputs,

O is the set of outputs;

and one operation:

T: *S* x I x O x *S* ----> *S* is the transition function: T(s,i,o,s') which maps the state s to state s' when the input and output actions i and o respectively, are observed to occur.

The transition function of the Transport protocol entity with four arguments given above, has a typical value given by: T(closed, TCONreq/s_CR, wait_for_CC), where the arguments represent the previous state, the input/output actions and the next state respectively. The slash between input and output arguments is syntactical, denoting that the input action is followed by the output action is uninterruptible during the state transition of the system. The transition is assumed to be atomic.

The set of all states, the input and output action labels and the set of transition values are illustrated in Figure 3.1. This is the familiar state transition diagram of an LTS.

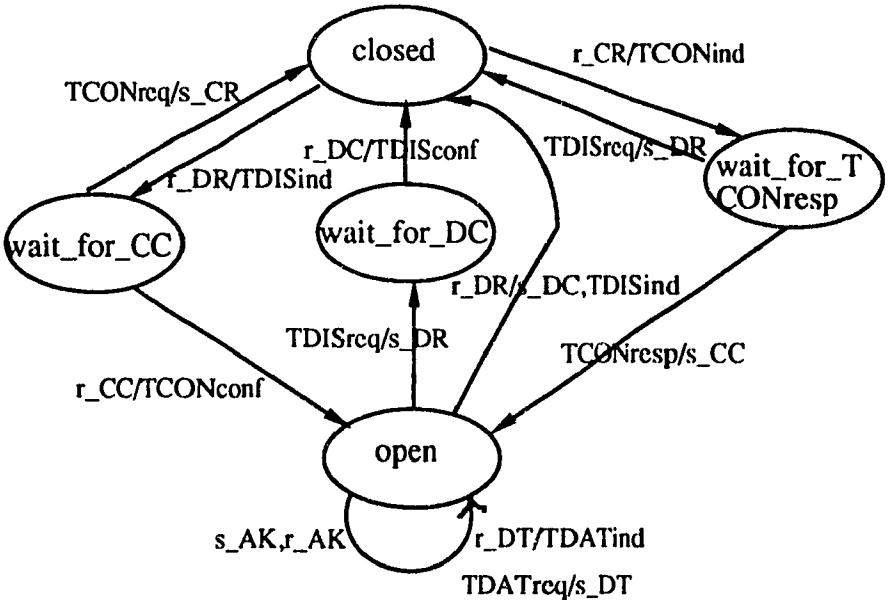


Figure 3.1. State transition diagram of the OSI Transport protocol (class 0)

Tables 3.1 and 3.2 list the states and input/output labels respectively, of the state transition diagram of Figure 3.1. The column of corresponding comments provides an informal explanation of what the states and labels are meant to represent in the system.

STATE	COMMENT
idle	Initial state of the system
wait_for_CC	Wait for a Connect Confirm primitive
open	A connection has been established and data can be exchanged
wait_for_TCONresp	Wait for a Transport Connect response
wait_for_DC	Wait for Disconnection

Table 3.1. State labels in an LTS description-Transport protocol

Transport

INPUT/OUTPUT LABEL	COMMENT
TCONreq(resp)	Transport connection request or response service primitive received from (sent by) session user.
TCONconf(ind)	Transport connection confirm or indication service primitive sent by (received from) session user.
TDISreq(ind)	Transport disconnect request or indication service primitive received from (sent to) session user.
TDATreq(ind)	Transport data request or indication service primitive received from (sent to) session user.
s(r)_CR	Send or receive a Connect request to (from) a peer protocol entity.
s(r)_CC	Send or receive(a Connect Confirm to (from) a peer protocol entity.
s(r)_DT	Send or receive a data (from) a peer protocol entity.
s(r)_Ak	Send or receive data Acknowledgement to (from) a peer protocol entity.
s(r)_DR	Send or receive a Disconnect request to (from) a peer protocol entity.
s(r)_DC	Send or receive a Disconnect Confirm to (from) protocol entity.

Table 3.2. Action Labels of an LTS

3.2 Process Algebraic Models

It is possible to unfold a state transition diagram into an acyclic linear graph in the form of a tree [55]. The linear tree represents the sequences of state transitions corresponding to the different paths identifiable in the transition diagram. The tree repeats itself after the first complete unfolding of the transition graph.

If the states at the nodes of such a tree are assumed to be unobservable by the environment, removal of state labels from the nodes of the tree, transforms the state transition model into a black box process model [54].

In the black box model of a CCS process, state transitions are unobservable by the environment and any of its agents. Actions performed on such a black box are experiments conducted on the process by the environment. The process to be described reacts non-deterministically to the experiment offers made by the environment. Synchronisation of the process with the environment for a given experiment results in the successful occurrence of an action, or deadlock.

System descriptions in such behaviour models take the form of algebraic equations of processes. Process behaviour is defined by expressions on observable actions. Internal actions, unobservable by the environment are modelled as internal non-deterministic behaviour of the process. Non-deterministic behaviour of the process and the environment is usually modelled as non-deterministic choices between alternate behaviours of the process. Such descriptions are called process algebras.

Recently, there has been an increasing interest in applying process algebraic specification techniques (CCS,CSP, ACP and LOTOS) to distributed systems.

Process algebras are labelled transition systems, which may include an action that is not in L, known as the silent action "τ" in [54], "I" in [57], and "i" in [25]. The actions "τ" and "i" may be interpreted as an internal step of the process which is unobservable or hidden from the environment. Such actions are used to model "internal" non-determinism in CCS and LOTOS. Non-determinism of observable actions is modelled by a choice

operator in CCS, ACP and LOTOS, denoted by '+', '+' and '[]' respectively. In CSP two kinds of observable non-determinism are identified. One type of non-determinism assumes no explicit participation of the environment in making the choice between alternate behaviours, by the use of the 'I' operator. The second kind of non-determinism defined in CSP, is one in which the environment can control the choice between alternate behaviours. Alternate behaviour in this case is expressed by using the general choice operator ([]).

In these systems (except in CSP) the set of action labels is given by $L' = L \cup \{i\}$ and the transition function T is defined over L'. In LOTOS $L' = L \cup \{i,\delta\}$ where δ is an internal action representing successful termination.

Behaviour descriptions are given in terms of behaviour expressions over the set of actions in L'. Transition functions in the form of axioms and inference rules for the derivation of behaviour provide the operational semantics of behaviour expressions.

Definition 3.1

A process algebra is a heterogenous algebra [S,B,L',T, Ω] with the carriers:

S is the set of states;

B is the set of behaviour expressions B defined over action labels and behaviour expressions;

L' is the set of action labels;

Ω is the set of operations on behaviour expression from β

nil ----> β

ω: β x L' x β ----> β

T: β x L' x β ----> β

T is the transition function. The transition function $T(B,\alpha,B')$ transforms B into B' with the occurrence of action α. It is defined by a set of axioms and inference rules

relating a behaviour expression B and action labels from L' to behaviour expressions. B is defined by operators from Ω.

[end of Definition 3.3]

Behaviour expressions define possible behaviours of the system. Inferred system behaviour depends on the form of the expression determined by the operators of Ω and the occurrence of actions.

To illustrate, Table 3.3 lists the type of expressions which can be formed by the operators Ω with behaviour expressions $E \varepsilon \beta$ and $\mu \varepsilon L'$ in CCS. The inference rules of the transition function T of CCS are given in Table 3.4.

Operators	Syntax	Remarks
Inaction	**NIL**	
Action prefix	**μ.E**	an action followed by an expression E
Choice	**E1 + E2**	a choice of behaviours E1 or E2
Composition	**E1|E2**	a composition of behaviours
Restriction	**E\μ**	a restriction on μ
Relabelling	**E[Φ]**	Φ: L' ----> L', a relabelling
Recursion	**recX.E**	recursion of the expression X in E, with X=E

Table 3.3. CCS expressions - Syntax

---

1. $\mu.E \xrightarrow{\mu} E$

2. $\dfrac{E1 \xrightarrow{\mu} E' \text{ or } E2 \xrightarrow{\mu} E'}{E1+E2 \longrightarrow E'}$

3. $\dfrac{E1 \xrightarrow{\mu} E1'}{E1|E2 \xrightarrow{\mu} E1'|E2} \qquad \dfrac{E2 \xrightarrow{\mu} E2'}{E1|E2 \xrightarrow{\mu} E1|E2'}$

$$\frac{E1 \xrightarrow{\mu} E1' \qquad E2 \xrightarrow{\bar{\mu}} E2'}{E1|E2 \xrightarrow{\tau} E1'|E2'}$$

4. $\dfrac{E \xrightarrow{\mu} E1' \qquad (\mu,\mu^* \notin A)}{E\backslash A \xrightarrow{\mu} E\backslash A}$

5. $\dfrac{E1 \xrightarrow{\mu} E1'}{E[\Phi] \xrightarrow{\Phi(\mu)} E'[\Phi]}$

6. $\dfrac{E1\{recX.E2\} \xrightarrow{\mu} E'}{recX.E2 \xrightarrow{\mu} E'}$

Note 1: μ,μ* are complementary actions in L' which can synchronise when two expressions are composed.
Note 2: recX.E2 is obtained by substituting the expression E2 for X simultaneously in all occurrences of X avoiding clashes of bound variables.

Table 3.4. Semantics of CCS

In Table 3.4 each inference rule is stated in the form of a transition which transforms one expression to another accompanying the occurrence of an action. The implication of the transition is given below the line. Given a behaviour expression and a transition on an action, an appropriate inference rule can be used as the basis for rewriting the behaviour expression.

3.3 Behaviour Expressions Of Processes

Behaviour expressions define abstract processes. The axioms and inference rules which form the transition function of a process algebra provide the operational semantics for the interpretation of behaviour expressions.

A behaviour expression B can be represented by an action tree. The root of an action tree represents the entire possible behaviour of a process P. Possible behaviours are derived from the axioms and inference rules of the transition function T. The trees thus obtained are called Derivation Trees (DTs).

Example 3.1

An example of a behaviour expression in CCS and its DT is given below.

Let a process P be defined by the expression:

$$P = \alpha.(\beta.NIL + \tau.\gamma.NIL) + \alpha.\gamma.NIL \qquad (3.1)$$

The entire behaviour of P can be inferred from its defining expression on the RHS of Equation (3.1).

Thus, we can infer P --α--> P1 and P--α-->P2, where P1= β.NIL + τ.γ.NIL and P2 = γ.NIL. Further, we can infer P1 --β--> NIL and P1 --τ--> P2, and P2 --γ-->Nil. Figure 3.2 shows the form of the DT for the behaviour expression for P.

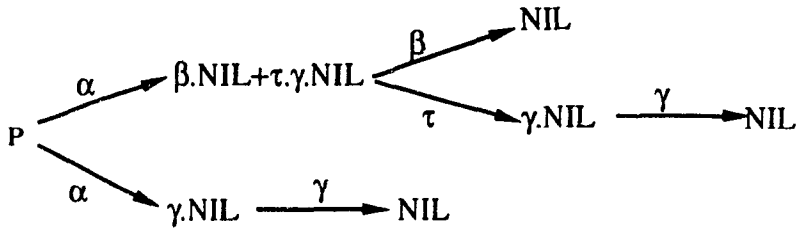


Figure 3.2. The derivation tree (DT) for the expression for P

Figure 3.3 shows the same DT with the node information omitted, and is called the action tree of P.

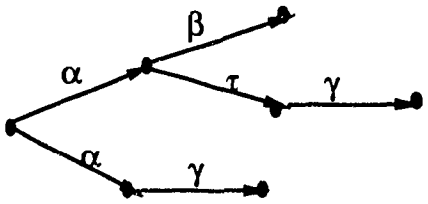


Figure 3.3. The action tree of P

[end of example 3.1]

A consequence of expressing internal non-determinism in terms of the internal action τ, is that the observed behaviour of two processes is "sometimes similar" implying

that they may be equivalent. For instance, the two action trees in Figure 3.4 behave

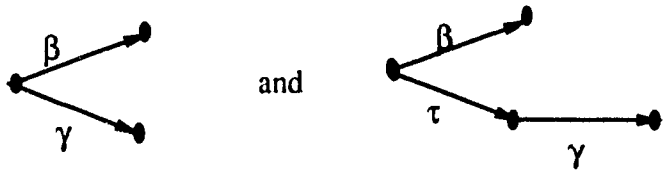


Figure 3.4. Deadlock due to τ–action in an action tree

"sometimes similarly".

An β or a γ experiment with the process represented by the leftmost tree of Figure 3.4 will always be successful, while that of the second will some times deadlock with a β experiment. The deadlock is due to the execution of a τ (an internal) action by the system, making it impossible for a β experiment to succeed.

The basic question which emerges about system behaviour from such experiments is the observational equivalence of processes.

3.4 Observational Equivalence

Observational equivalence of process behaviour is based on the idea that system behaviour is determined by the way it interacts with external observers. "It defines a monotonically increasing set, upto a maximum fixed point, of equivalence relations between states of two specifications" [55].

Intuitively, two processes are considered to be "observationally equivalent if they are indistinguishable by experiments of an observer. That means two such behaviours may simulate each other when interacting with an experimenter, in a manner such that

their observable behaviour is indistinguishable. As a result an observer may conclude that the two expressions are 'equivalent'.

Interpretations of equivalence are provided by the theory of equivalence. Equivalence classes for weak bisimulation equivalence, congruence [54],[55], testing equivalence [60] determine whether given behaviour expressions belong to the same equivalence class. The method of achieving this determination involves the use of transformation laws based on axioms and inference rules of the process algebra under consideration.

Two fundamental notions of observational equivalence defined in [54],[55] are called 'strong observational equivalence' and 'weak observational equivalence'. The difference between them is that of granularity of the equivalence relation. For given states of two processes, strongly equivalent processes simulate each other step-by-step for a given action sequence, while weakly equivalent processes simulate each other for transitions, with respect to a given action sequence, from equivalent states to equivalent states with possible intervening τ actions. The notion of congruent processes results from strongly equivalent processes with every state of one being equivalent to a corresponding state of the other.

3.4.1 Definitions Of Observational Equivalence

For an LTS <*S*,L,T,s0>, the formal definitions characterising strong and weak equivalences based on [55] are given below.

Let the integer K index the set *S* of an LTS and define a function:

F: 2^{KxK} ----> 2^{KxK} as follows:

Definition 3.4

Let R be a binary relation over K. Then:

$F(R) = [(p,q) | p,q \in K$ and for every $\mu \in \Sigma$

(i) if $p \xrightarrow{\mu} p'$ then ($\exists$ q': $q \xrightarrow{\mu} q'$ and p' R q')

(ii) if $q \xrightarrow{\mu} q''$ then ($\exists$ p": $p \xrightarrow{\mu} p''$ and p" R q")]

Function F possesses the following properties (proofs are in [55]:

Proposition 3.4

(a) $R1 \leq R2 \Longrightarrow F(R1) \leq F(R2)$.

(b) If R is an equivalence relation, then so is F(R).

[end of Definition 3.4]

Strong equivalence is defined inductively as follows:

Definition 3.5

The binary relations $\sim_k$, $k \geq 0$, are given inductively as follows:

(a) $\sim_0 = K \times K$

(b) $\sim_{k+1} = F(\sim_k)$, $k \geq 0$.

If $p \sim_k q$ we say that p and q are k-strongly equivalent.

[end of Definition 3.5]

It can be shown on the basis of Proposition 3.4 that the relation $\sim_{k+1}$ is a refinement of $\sim_k$, i.e.$\sim_{(k+1)}$ is included in $\sim_k$, $k \geq 0$.

Definition 3.6

States p and q are strongly equivalent, written $p \sim_\omega q$, if $p \sim_k q$ for every $k \geq 0$. That is:

$(\sim_\omega) = \cap(\sim_k) \quad k \geq 0.$

[end of Definition 3.6]

It is also easy to show that $\sim_k$, $k \geq 0$ and $\sim_\omega$ are true equivalence relations over transitions of system states.

Weak Observational Equivalence is similarly defined as follows:

The function F: $2^{K \times K}$ ----> $2^{K \times K}$ is modified and denoted by G to account for τ-actions.:

Definition 3.7

Let R be a binary relation over K. Then:

G(R) = {(p,q)| p,q $\in$ K and for every s $\in \Sigma^*$

(i) if p =s=> p' then (q': q =s=> q' and p' R q')

(ii) if q =s=> q" then (p": p =s=>p" and p" R q")}

Function G can be strictly smaller than, equal to, or strictly larger than R. Also proposition 3.4 holds for G as well as F.

[end of Definition 3.7]

Weak observational equivalence, or simply "observational equivalence" is defined inductively as follows:

Definition 3.8

The binary relations $\approx_k$, $k \geq 0$, are given inductively as follows:

(a) $\approx_0 = K \times K$

(b) $\approx_{(k+1)} = F(\approx_k)$, $k \geq 0$.

If $p \approx_{(k+1)} q$ we say that p and q are k-observationally equivalent.

[end of Definition 3.8]

Definition 3.9

States p and q are observationally equivalent written $p \approx_\omega q$, if $p \approx_k q$ for every $k \geq 0$. That is:

$\approx_\omega = \cap \approx_k \quad k \geq 0.$

[end of Definition 3.9]

The notions of strong and weak observational equivalences can also be defined in terms of weak and strong bisimulation relations [55].

All the above equivalences are defined in the environment of a CCS context given by:

Definition 3.10

A CCS context C[ ] is a CCS expression where some sub-expressions have been replaced by "holes".

[end of Definition 3.10]

Definition 3.11

We say that CCS expressions E1 and E2 are observationally congruent (have the same meaning), written $E1 \approx^c E2$, iff for all contexts C[ ], $C[E1] \approx C[E2]$;

Where C[E] represents the expression obtained by filling the holes in C[ ] with expression E.

[end of Definition 3.11]

It turns out that $E1 \approx^c E1'$ and $E2 \approx^c E2'$, then $E1 \approx E2$ iff $E1' \approx E2'$.

In other words the observational equivalence of two expressions E1 and E2 is not affected when they are algebraically transformed without leaving their respective congruence classes. The main reason for introducing $\approx^c$ is to allow such safe syntactical manipulations of behaviour expressions.

In principle, in order to obtain a state transition diagram which is equivalent to a given behaviour expression, the entire expression B0 defining the process P is mapped to the initial state s0 of the corresponding state machine. Then each transition: $B \text{ --}\mu\text{-->} B'$ of expressions is mapped to a corresponding state transition of the form: $s \text{--}\mu\text{--> } s'$.

This is difficult to do in the case of infinite expressions. Algorithms such as the unfolding algorithm [55] and other such algorithms surveyed in [12] exist for mapping finite DTs to state transition diagrams.

CSP is similar to CCS but uses trace equivalences and refusal sets to prove logical properties of the system. Also, there are some differences in modelling non-determinism and synchronisation semantics of CCS and CSP.

3.5 Initial Algebra Semantics And Behaviour

ACP provides a unified view of existing process algebras by using an axiomatic model. In the axiomatic approach used in ACP, axioms on the operations Ω of Definition 3.2 are intended to model notions required to define the distributed behaviour of processes.

Thus, the set of operations include operators which model non-deterministic behaviour in terms of the alternative composition operator (+), and concurrent and communication behaviour in terms of the merge (or parallel composition) operators (||, ⌊⌊ and |). The possible behaviours are terms generated by the application of terms in Ω. The set of axioms, define relationships between the operators which induce an initial algebra semantics in to ACP behaviour specifications.

Example 3.2

As an illustrative example a data type specifying the initial algebra of natural numbers is given below, in the notation of ACTONE [44].

type nat_numbers
sort nat
opns **0**: ---> nat
succ: nat ----> nat
\+ : nat,nat ---> nat
eqns For all **x,y** nat:
$x + 0 = x$
$succ^n(0) + succ^m(0) = succ^{(n+m)}(0)$
$x + succ(y) = succ(x+y)$
[end of example 3.2]

In this example, the set of operations are **0**, **succ** and + , and their essential properties are given in terms of three equations. The superscripts **n** and **m** are integers denoting the successive application of the **succ** operation **n** and **m** times respectively. These three equations are the axioms of the algebraic system of natural numbers which induce initial algebra semantics into the specifications.

Example 3.3

The set of operations used in ACP are given in Table 3.5 below:

OPERATOR SYMBOL	COMMENT
+	alternative composition (sum)
.	sequential composition (product)
| |	parallel composition (merge)
| |_	left merge
|	communication merge
∂_H	encapsulation
τ_I	abstraction
δ	deadlock (failure)
τ	silent (internal) action

Table 3.5. The set of operators Ω of ACP

In relation to CCS, the operators +, ||, and τ have exactly the same meaning; multiplication is more general than the prefix multiplication of CCS; | |_ and | correspond to partially to the interleaving operator and the notion of parallel synchronisation. δ is similar to **NIL** in sums (but not in products). ∂_H used for hiding the set of actions H from the environment, so that it cannot communicate with the environment. And τ_I in CCS (*i* in LOTOS) is used to abstract from internal actions used in an expression.

The initial algebra semantics of ACP is given by the set of axioms given in Table 3.6

$x + y = y + x$	A1
$x + (y+z) = (x+y) + z$	A2
$x + x = x$	A3
$(x + y).z = x.z + y.z$	A4
$(x.y).z = x.(y.z)$	A5
$x + d = x$	A6
$d.x = d$	A7
$a\|b = b\|a$	C1
$(a\|b)\|c = a\|(b\|c)$	C2
$d\|a = d$	C3
$x\|\|y = x \|\|\_ y + y \|\|\_ x + x\|y$	CM1
$a \|\|\_ x = a.x$	CM2
$(ax) \|\|\_ y = a(x\|\|y)$	CM3
$(x + y) \|\|\_ z = x \|\|\_ z + y \|\|\_ z$	CM4
$(ax)\|b = (a\|b).x$	CM5
$a\|(bx) = (a\|b).x$	CM6
$(ax)\|(by) = (a\|b).(x \|\| y)$	CM7
$(x + y)\|z = x\|z + y\|z$	CM8
$x\|(y + z) = x\|y + x\|z$	CM9
$\partial H(a) = a$ if $a \in H$	D1
$\partial H(a) = d$ if $a \in H$	D2
$\partial H(x + y) = \partial H(x) + \partial H(y)$ if $a \in H$	D3
$\partial H(x.y) = \partial H(x).\partial H(y)$ if $a \in H$	D4

Table 3.6. Axioms of ACP

In these equations:

(i) $a,b,c,....\varepsilon$ A, the set of atomic actions (also called 'steps' or 'events'). A is also referred to as the alphabet. A is assumed to be finite.

(ii) x,y,z,... are variables, ranging over the domains of processes (process algebras).

[end of example 3.3]

One useful property which results from a behaviour definition based on the axioms of ACP is that of a system of guarded recursive equations:

If $X = \{X_1,...,X_n\}$ is a set of labels of formal process variables, and a system E_X of guarded fixed point equations (or guarded recursion equations) for X is a set of n

equations $\{ X_i = T_i\{X_1, ...,X_n) \mid i = 1,...,n\}$ with $T_i(x)$ a guarded term then the following theorem has been proved in [7]:

Theorem : Each system E_X of guarded fixed point equations has a unique solution in $(A^{\infty})^n$.

ACP is extended to ACP_τ by introducing additional axioms related to the abstraction mechanism.

Issues related to proving correct system behaviour are discussed in Chapter 5.

"When I use a word", Humpty dumpty said, in a rather scornful tone, "it means just what I choose it to mean - neither more nor less."
"the question is" said Alice, " whether you can make words mean so many different things"
- Lewis Carroll in 'Through the Looking Glass'

Chapter 4

DISTRIBUTED SYSTEMS-SERVICES AND CONSTRAINTS

In this chapter, services provided by distributed systems are considered. The "service" concept naturally leads to the consideration of requirement descriptions. The new notion of partial constraints is introduced in this chapter. The idea of partial constraints evolves naturally from requirement descriptions. The design methodology based on partial constraints is then introduced by means of several examples. Partial constraints are demonstrated to be useful conceptual tools in system design.

4.1 Service Requirements And Architectures

Providing distributed services to users is the primary objective of distributed system design. The twin concepts of computer system architecture and services provided to users emerged with Operating system theory [32] in computer science and was refined in software engineering (see for example:[65]) Integration of communication networks with computer systems carried over these concepts to the area of distributed services and architectures - one defining the other. Any description of distributed systems services subsumes system architecture and a description of architecture assumes system services. This in turn implies that the level of abstractness and formality of description of each - complements the other. For example, in [70] communication services are described in terms of abstract machines. That means an implementation independant description.

The layered architecture [86] of the OSI reference model define services in terms of interactions at layer boundaries. Layer boundaries are abstract interfaces between service provider and service user. Interactions between user and provider occur in the form of abstract service primitives executed at abstract interfaces. In Figure 4.1 interactions between user and provider are shown by double-headed arrows at layer interfaces. Abstract service primitives are parameterised operations by means of which communication services are requested and provided. In ISO documents the abstract service primitives and abstract interfaces are called ASPs and SAPs (Service Access Points) respectively (cf. [40]). ASPs are operations executed by users and the service machine at the SAPs. The system resulting from the composition of the protocol entities and the underlying service is called the service machine. Transfer of data between user entities and underlying protocol entities is also accomplished by the use of ASPs.

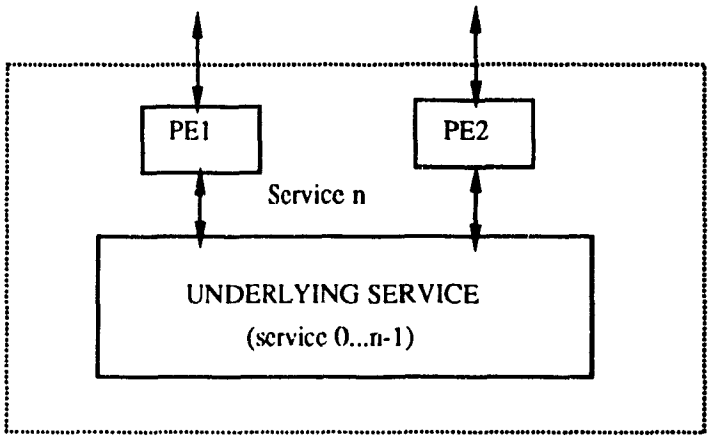


Figure 4.1. Interactions at service access points

Services and protocol descriptions contained in ISO standards documentation are informal and therefore prone to ambiguities. The abstract service machine is interpreted to be an abstraction of the protocol and the underlying services it uses. The need for formal

notions of an abstract service machine and its environment are an immediate consequence of informal descriptions.

4.1.1 Formal Notions Of Service

The importance of the service concept in OSI systems and its relation to the service machine was pioneered in [79]. In [80], the idea that abstract service definitions be independant of protocol descriptions was proposed. Standardisation of formal service descriptions based on system architecture was also proposed. The provision of verification proofs of service obtained from protocol/underlying-service combinations in the appropriate standards was recommended.

Informal, or semi-formal, descriptions make system design and analysis difficult. Standardisation of the FDTs, LOTOS, ESTELLE and SDL now point the way to standardised formal service, as well as, protocol specifications. In principle, formally specified standards should facilitate verification and validation proofs.

4.2 Service Specifications

Formal specification of system behaviour presumes a behaviour model and a design methodology. Many different models of behaviour used for the specification of OSI system services are described in the literature. The principal models of interest are the Abstract Data Type model [75],[13],[45], the finite state machine model [11] and the process algebraic model [70].

4.2.1 Algebraic Type Specifications

The essential feature of the algebraic type specification approach is apparent from the syntactic structure of the abstract data type specification given in Example 3.5.1. The set of user service primitives are specified as the signature of a type, with service properties expressed by the semantical (or equational) part of the specification. The user operation interface is abstract. Logical properties of a service can be derived from the equational part of a type specification. A type specification may abstract from an implementation architecture. For example, the Last-In-First-Out (LIFO) service properties of a stack can be derived from the equational part of a stack specification. An implementation of a stack may consist of a list structure constrained to provide the LIFO behaviour of a stack. On the other hand, a structure such as a list or a queue may also be specified.

In [76] for example, the Alternating Bit protocol and Stennings data transfer protocol [74] are specified as a combination of ADTs and state transition systems. System architecture is implicitly assumed to be that of a queue connecting two users. User service primitives include the send (add) and receive (remove) operations on the queue. Data is transferred as arguments of the send and receive operations on the queue. Service properties expressed in the type equations relate user operations to the state of the queue.

The OSI transport service specified in [46] is wholly a data type specification. The set of SAPs and the system are modelled as list and queue structures respectively. Connection endpoints are modelled as tuples of predefined data type values and an enumeration type is defined to model states of the system. Equations relate service operations to each other from which allowed sequences of interactions (or traces) can be directly derived.

4.2.2 Finite State Machine Based Specifications

In [9], the OSI transport service specification results in a state transition diagram very similar to that provided in the standards. However, state transitions which are instantaneous and atomic do not accurately model the finite computations which accompany such transitions. Additionally, two distinct FSMs simultaneously making transitions (assumed to be synchronised) at the SAPs, considerably weaken the expressive and analytical power of the model of specification. The global state space explosion under composition is a well-known problem of state transition descriptions. However, strategies of reachability analysis of FSMs demonstrated in several papers (cf. for example [86]) provide for a satisfactory verification technique.

4.2.3 Process Algebraic Specifications

The process algebraic model based FDT LOTOS is used to describe session services and transport services in [19] and [65] respectively. In these specifications the architectural framework is that of the OSI reference model. The transport service, for example, is modelled as a process with a single LOTOS event gate - the transport SAP, interfacing the process with the environment. The environment consists of session users. Service is described in terms of a behaviour expression relating the possible interactions of the provider (service) process with that of session user processes. However, verified transport and session service specifications do not exist at the time of writing this thesis.

4.3 Service Design And Specification

The examples cited above specify a communication service. In general, a clear relationship between architecture and service requirements is difficult to deduce from

specifications. That means that the specification does not directly provide a means to verify whether system architecture and specification is consistent with service requirements. An analysis of the relationship between architecture and service requirements would be helpful in the design process.

4.3.1 Constraints And Design

In this section, the notion of constraints and a design method based on constraints, for distributed system design is proposed.

The principal guide to system design in all engineering disciplines is the set of constraints which provide the limits for the dynamic behaviour of the system under design. Constraints on system behaviour are obtained from requirements and the fundamental characteristics of system behaviour. The fundamental characteristics of distributed system behaviour are (Chapter 1): Non-determinism, Concurrency (Sequentiality), Communication (Synchronisation), Composability and Distributability.

As an introduction to the evolution of the idea of constraints, a brief description of constraint-oriented specifications is given below.

4.3.2 Constraint-oriented Specifications

The "divide-and-conquer" approach to design, in the context of traditional programming languages allows the programmer to structure his program. Similarly, the constraint-oriented style permits the separation of system concerns into a collection of separately defined smaller processes which together, are intended to specify system behaviour. Each process is presumed to define a separate concern or 'constraint' of the system. Though only observable interactions of processes are represented in a constraint-oriented specification, their temporal ordering relationships are composed together as the

conjunction of separate constraints. Each constraint is defined on the relevant subset of the set of interactions being specified.

A simple example taken from [10] illustrates the 'constraint-oriented' approach:

Example 4.1

Given a process P defined by the LOTOS behaviour expression:

P := a ; b ; c ; stop [ ] b ; a ; c; stop (4.1)

where the operator '[]' denotes a non-deterministic choice, we may say that P satisfies the constraints:

CO1:

"a precedes b precedes c precedes inaction"

or

"b precedes a precedes c precedes inaction" (4.2)

Where a,b,c belong to the alphabet of events associated with the behaviour defined in Equation 4.1 and the null event stop is inaction. The 'or' in CO1 is presumed to be non-deterministic, in the sense that an observer in the environment records either of the traces <a,b,c> or <b,a,c>, the choice being made autonomously by the process P.

A trace [35], is the observed behaviour of a process in terms of a finite sequence of symbols, recording events in which the process has engaged in, upto some moment in time. The set of traces {<a,b,c> and <b,a,c>} is denoted as Tr(P), where P is a process

defined by an algebraic behaviour expression as in Equation 4.1. The set Tr contains all possible observable traces of the process.

If the action c in both parts of the expression of Equation 4.1 is unique and synchronisable, then Equation 4.1 can be rewritten as Equation 4.3 by using the inference rules of LOTOS, in terms of the parallel (||) operator of LOTOS:

P := a ; c ; stop |c| b ; c ; stop (4.3)

Axioms and inference rules of LOTOS may be used to rewrite given behaviour expressions into equivalent expressions by repeated replacement modulo a desired level of equivalence.

Inversely, Equation 4.3 can be reduced to Equation 4.1 using the expansion theorem for parallel expressions in CCS. Behavioural equivalence of the two expressions implies that their trace sets are the same. The converse however is not necessarily true. Equivalent trace sets do not necessarily imply observational equivalence.

An example of two expressions which are trace equivalent but not observationally equivalent is:

P1:= i ; (a ; stop [] b ; stop)

and

P2 := i ; a ; stop [ ] i ; b ; stop (4.4)

where the internal event is represented by "i". Here, an experiment offer of "a" *or* "b" by the environment could result in deadlock with P2 but not with P1, even though their trace sets are equal. Deadlock in the case of P2 is due to 'i', which may cause the process to 'silently' (unobservably) slip into a state where it offers either the experiment 'a' *or* 'b' but not both. This is not the case for the 'i'-step in Equation 4.4.

The process of Equation 4.3 offers a and b in either order followed by c, where c* denotes synchronisation of the event-offers c, common to both parts of the behaviour. The notation "c*" has been used in an ad-hoc manner.

The expression for P in Equation 4.3 may again be presumed to define the constraint:

CO2:

"a precedes c* " *and* " b precedes c* " (4.5)

The conjunction of these two components gives the constraint-oriented specification of the behaviour in Equation 4.3.

The behaviour expressions of Equation 4.1 and Equation 4.3 have the same trace set - <a,b,c*> and <b,a,c*>

A further constraint such as "x precedes c" can be added later to this behaviour expression thus:

P := (a ; c ; stop |cl b ; c ; stop) |cl x ; c ; stop (4.6)

The expressions in Equation 4.3 and Equation 4.6 are a conjunction of interdependant constraints. If the action "c" is deleted from Equation 4.3 the parallel (||) operator may be replaced by the interleaving (|||) operator to represent the independence of each component of the corresponding constraint - CO2

[end of example 4.1]

4.3.3 Constraints

System design is based on statements of requirements. Intuition of system behaviour based on service architecture and functionalities of system components help in the design of the system. Intuitively, the idea of constraint oriented design is to consider assertions on system behaviour as constraints or limits within which system behaviour can be specified. An aid to the design process is the identification of possible orders on sets of related communication events which satisfy functional and architectural requirements.

Viewed in this way, system behaviour is circumscribed by constraints. For example, a constraint may be constructed to derive a behaviour specification such as the one in Equation 4.3. Conversely, given a behaviour specification like that of Equation 4.3 it should be possible to show that it satisfies the constraint CO2.

In what follows, constraints are assertions relating events of a communication environment, in which the system participates. Related constraints can be combined to yield again - constraints. That means the system of constraints is closed with respect to the relations between events.

A consequence of Definition 3.1.2 of the previous chapter is that an LTS is an algebra represented by the tuple <S,L,T,s0>, accordingly a distributed system is defined as follows:

Definition 4.1 A Distributed System

A distributed system is an LTS - S, such that a decomposition of S into component systems P1, P2 and uS exists, with:

$S = P1 \times P2 \times uS$

where P1, P2 and uS are represented by the algebras <P1,L1,T1,p10>, <P2,L2,T2,p20> and <uS,uL,uT,us0> respectively. The LTSs satisfy the following conditions:

(i) s0 = [p10,p20,us0]

(ii) $P1 \cap P2 = \Phi$ and $L1 \cap L2 = \Phi$; where Φ is the empty set

(iii) $L1 \cap uL \neq \Phi$ and $L2 \cap uL \neq \Phi$.

[end of Definition 4.1]

Condition (i) simply ensures a unique initial state in the event that it is possible to obtain a consistent combination of states of the components of the system. Condition (ii) states that P1 and P2 have no common states or interactions, while condition (iii) states that both L1 and L2 have common actions (interactions) with uL. Distributed architecture is implicit in conditions (ii) and (iii) in that P1 and P2 represent independant distributed components, with no direct interactions, and uL the non-distributed component of the distributed system S. The non-distributed component uL is capable of independant as well as joint actions with P1 and P2. The definition given above may be extended to allow for the decomposition process to result in many components in the usual manner.

Partial Constraints are introduced with an example from [55].

Example 4.2

Let processes P1 and P2 be defined by the behaviour expressions:

$$P1 = \alpha.(\alpha.(\beta + \gamma) + \alpha.\beta + \alpha.\gamma)$$

and

$$P2 = \alpha.(\alpha.(\beta + \gamma)) + \alpha.(\alpha.\beta + \alpha.\gamma) \qquad (4.7)$$

The action trees of P1 and P2 are given in Figure 4.2.

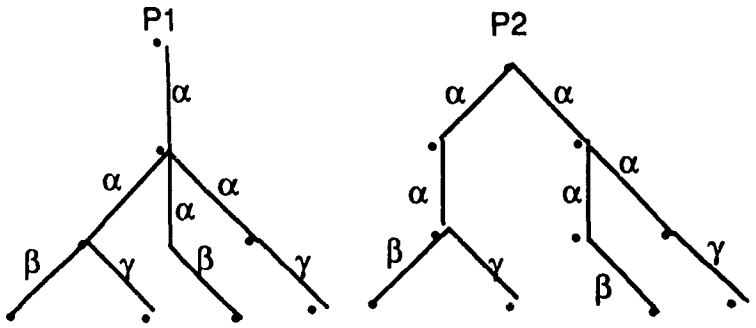


Figure 4.2. Action trees of P1 and P2

[end of example 4.2]

CCS action trees grow as associated communication events occur. Action tree growth is linear according as new actions occur in temporal relation to previous actions. Branching or non-linear growth occurs, if a new action is non-deterministically related to previous ones.

Constraints can be constructed from trees as follows. Linear growth of an action tree along a path of the tree is related by a precedence relation on actions. Branching growth corresponding to non-deterministic choices between actions is related by the non-deterministic 'or' relation.

For the action trees of Figure 4.2, the behaviours of processes P1 and P2 respectively must satisfy the constraints:

C03:

α precedes

((α precedes (β *or* γ))

or

(α precedes β)

or

(α precedes γ)) (4.8)

C04:

(α precedes (α precedes (β *or* γ))

or

(α precedes (α precedes β) *or* (α precedes γ)) (4.9)

Constraints C03 and C04 have been constructed stepwise (action by action) with a precedence relation for new actions in a given path and the choice relation (non-deterministic '*or*' relation) for actions on different branches. Actions in constraints are self-referential.

But distributed system behaviour manifests two other aspects. Communication between actions in an environment - an aspect of non-determinism, and system architecture. Non-determinism is discussed in the next section while system architecture is treated in the next chapter.

4.3.4 Non-determinism, Communication And The Environment

For example, CCS actions represent communication between a process and its peers in the environment, synchronising on matching event(experiment)-offers. Thus traces represent sequences of successful communication events (actions) of the process under definition with respect to the environment. Reducing a specification to behaviour expressions consisting of sequential terms only represents satisfaction of precedence constraints, provided non-determinism does not alter the behaviour properties of the system.

Indeed, precedence relations on a given set of actions in an environment defines classes of behaviours which are trace equivalent. Each trace satisfies the logical property represented by the sequence of precedence related actions in a constraint. Examples of traces satisfying logical properties represented by precedence relations on actions in a constraint are given in Chapter 5.

Thus for example, the properties:

PR01:

α precedes α precedes β (4.10)

and

PR02:

α precedes α precedes γ (4.11)

are satisfied by both P1 and P2 of example 4.2.

But the property stated as:

"An α-action is possible such that after any further α action it is necessary that both a β-action and a γ-action are possible"

and expressed as:

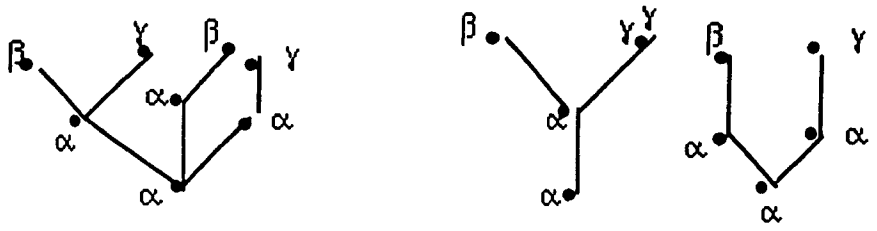
PR03:

$\exists\ \alpha$ s.t. ('select an α s.t.')

not (α precedes (α precedes (β *or* γ)))

is satisfied by P2 but not by P1.

Satisfaction of PR03 by P2 is proven in terms of partial order diagrams shown in Figure 4.3.



Poset of P1 Posets of P2

Figure 4.3. Poset diagrams for P1 and P2

The posets of P1 and P2 of Figure 4.3 are obtained as follows:

The occurrence of an action is represented by labelled nodes (dots) going bottom up. Nodes represent corresponding action edges of the DT. Branching is represented by edges in the poset leading to different nodes at the same level of the poset. Initial branching yields a forest of posets as is the case in the poset of P2 in Figure 4.3.

Thus the poset of a DT grows bottom up, sequential edges or branching edges leading from action to action. An initial choice between α actions is represented in the poset of P2 as the beginning of two different chains starting with the same action, each growing bottom up autonomously.

Assuming that the posets of Figure 4.3 have been obtained from the DTs of P1 and P2, satisfaction of PR03 by P2 is proved as follows:

Proof: (Satisfaction of PRO3 by process P2)

The posets of P1 and P2 show that initially, P2 can choose the action α of the second alternative poset (chain) in the poset forest of P2. This choice implies that it cannot choose any further α action whereby both a β action and a γ action are possible. The poset of P1 shows it cannot make such a choice of α actions initially, since it must choose the only α action offered. Therefore, PR03 is satisfied by P2 but not by P1.

[end of proof]

Alternatively, examination of C04 shows that property PR03 can be derived from C04 by selecting the second alternative component of C04. This implies that if the behaviour of P2 is derived from the requirement constraint C04, P2 will satisfy the logical properties of C04 as expressed by the component constraints of C04, but not C03.

It follows that PR03 also distinguishes P1 from P2. That means P1 and P2 are not observationally equivalent. But is that true in every sense? The answer is no! PR01 and PR02 are satisfied by both, but PR03 is not, which implies they are trace equivalent, but not observationally equivalent.

Indeed, P1 and P2 are k-observationally equivalent (k=2). K-equivalence implies observationally equivalence upto depth two of the DT. The 2-equivalence can also be seen in the precedence constraint description upto depth two. If the depth of the constraint is assumed to be the depth of its logical formula. The depth of a logical formula is the depth of the tree obtained by writing the expression in prefix notation. But P1 and P2 are not observationally equivalent to depth three of the tree. This is because of non-determinism, represented by branching in the DT going 'top-down', and going 'bottom-up' in the poset.

The non-observational equivalence of P1 and P2 shows that process behaviour cannot be captured solely by the satisfaction of precedence constraints. For example, P2 satisfies the non-deterministic constraint 'α *or* α' initially, but either choice leads to different chains. Also, deadlock and livelock properties - a consequence of non-determinism cannot be captured using trace semantics. Equation 4.3 is an example of non-satisfaction of deadlock properties in the presence of trace equivalence of two processes. The obvious conclusion is that trace semantics do not provide a sufficient set of tools for system design.

4.4 Constraint Relations

Indeed, a stronger result can be derived by examining typical DTs and posets, such as those of Figure 4.2 and Figure 4.3. It is shown in the sequel, that the non-deterministic relation between actions cannot be adequately captured only by partial orders on communication events.

In the poset representation of process DTs, the logical 'or' used in the constraints implicitly models non-determinism. But first, what are communication actions?

Recall that in the CCS model of process communication, the actions in Figures 4.2 and 4.3 represent synchronisation of event offers between the process under definition and other processes of the environment. In CSP matching input/output action value-offers of different processes result in communication with exchange (by input and output) of values. In event algebras [18] a synchronisation event of different histories of an environment result in a communication move (or action). A history can be viewed as a computational structure.

Indeed, we can assume that communication events are modelled by a form of action sharing as in ACP. This means that processes in an environment share an action (or actions) to communicate.

In this and the following chapter, we define a communication environment in terms of algebraic relations and formally define partial constraints on communication behaviour. The notion of partial constraints forms the basic notion on which a set of conceptual design tools is developed in this thesis. Illustrative examples are used to

demonstrate the use of design tools with respect to the first five design objectives stated in Chapter 1.

In what follows it is assumed that communication is an action (or a communication event), resulting from synchronisation or an interaction between actions. A computation due to communication is the result of an interaction. Communication between processes may result in a definable computation. A computational system has a structure, both temporal and architectural.

The above notions are elaborated below by using concepts of modal logic [39]. The four modal notions are that of 'necessity', 'impossibility', 'contingency' and 'possibility'. A proposition that is bound to be true is called 'necessary'; one that is bound to be false is termed 'impossible'. A 'contingent' proposition is neither necessary nor 'impossible'. A proposition that is not impossible is 'possible'. The phrase 'bound to be true' can be interpreted in different ways.

In order to elaborate the partial constraint design methodology, we proceed as follows:

1. Given an environment E of communication events called actions, a set of relations called partial constraint relations are defined on the actions of E. The actions of E are assumed to be self-referential.

2. Behaviours in E are functions defined on the actions of E. Assertions called partial constraints obtained from partial constraint relations on the actions of E, are derivable from behaviours by a definition of provability of the expression (satisfaction relation), B |= pc. Where B is a behaviour expression and pc is a partial constraint on the

actions in E. B |= pc can be viewed as a boolean, where B |= pc ==> that pc is true for B and B |= not pc ==> pc is false for B.

3. A method of constructing partial constraints for system specification is given.

4. The satisfaction relation |= between partial constraints and process algebras is defined providing a mapping from partial constraints to process algebraic behaviour expressions.

5. For verification purposes, the satisfaction relation |= is interpreted in terms of the modal operators □ ('henceforth') and ◊ ('eventually') are defined such that:

(i) If B |= □pc ≡ ∀ B' such that B==>B', then B'|=pc

(ii) If B |= ◊pc ≡ ∃ a B1,B2 (behaviour expressions)

and pc1,pc2 (partial constraints) in E

such that (B1 'op' B2)==>B and (pc1 'o' pc2) ==> pc ,

and B1|=pc1 and B2 |=pc2.

Where 'op' is an operator of the process algebra under consideration and 'o' is a relational operator for partial constraints defined below.

The operator □ is similar to the temporal logic operator 'henceforth'. It implies that pc is "henceforth" (always) true for all transformations of B, via interactions (or transitions). And the operator ◊ corresponds to the temporal logic operator 'eventually'. It implies that pc will "eventually" be true for a transformation (B1 'op' B2)==>B.

6. A method for validation and verification of specification is elaborated based on the modal operators defined as in step 4 above.

Steps 1,2 and 3 are described in this and the next chapter, while 4 and 5 are discussed in Chapters 5 and 6.

Assumption 4.1

If E is an environment consisting of communicating actions, then there exists a modality of determination (a systematic process of determination is modally 'possible') by means of which behaviours and related assertions on behaviours of communication systems can be determined.

If B is a set of behaviours and Pc a set of partial constraints on behaviours in E, then the assumption implies that:

(i) $\exists$ a method of determining partial constraints pc and behaviours B in B

(ii)..$\forall$ B ε B $\exists$ a pc ε Pc, s.t. B $\models$ pc.

[end of assumption 4.1]

Example 4.1

Examples of methods of modal determinations are:

(i) Given a communication environment E, a method of determining partial constraints or behaviours could be based on observable traces of system behaviour.

(ii) A system of canonical transformations on informal requirement descriptions of behaviours in E could be the basis of modal determinations.

(iii) Observed behaviour resulting from testing experiments could also be the basis of determinations.

An example of a definition of provability is any formal definition of the satisfaction relation B$\models$pc

[end of example 4.1]

Assumption 4.2

The partial constraint operators: 'or', 'and', 'not', $\forall$ and $\exists$, defined by the definitions given below, relate the actions of an environment E (defined below) and properly extend the semantics of the corresponding operators of First Order Predicate Logic (FOPL).

This implies that these operators can also be manipulated as FOPL operators in the elaboration of proofs of properties of behaviour specifications derived from partial constraints.

A similar remark applies to the operator 'precedes' which corresponds to the mathematical operator "less than" (<).

[end of assumption 4.2]

Definition 4.1 An Environment E

An environment E is a set of atomic actions labelled by a finite set of labels $\{\alpha,\beta,\gamma,...\}$, in which it is possible to encode a set of behaviours B on the actions of E, and a set of assertions called partial constraints (a set Pc), relating the actions of E. As per assumption 4.1, a definition of provability exists s.t. partial constraints are derivable from behaviours on E. i.e.

$\forall B \in \beta, \exists a\ pc \in Pc$, s.t. $B \models pc$,

the boolean function $B \models pc$ means that pc satisfies the behaviour B.

[end of Definition 4.1]

Partial constraints are defined next.

Definition 4.2 Partial Constraints

The set Pc contains the following well formed formulae (wff) on the actions of an environment E and the partial constraint operators '*or*', 'precedes', '*and*', 'not', '$\forall$' and '$\exists$'. They are called partial constraints and denoted by pc:

$\forall \alpha,\beta \in E$

(i) ' ' (the empty pc' is a wff

(ii) 'α', 'α *or* α','α *or* α *or* α...' are wff

(iii) 'α *or* β' is a wff

(iv) 'α precedes α' is a wff

(v) 'α precedes β' is a wff

(vi) 'α *and* α' is a wff

(vii)'α *and* β' is a wff

(viii) If pc is wff then so is 'α *or* pc'

(ix) If pc is wff then so is 'α precedes pc'

(x) If pc is a wff then so is 'α *and* pc'

(xi) If pc is a wff then so is 'not pc' is a wff

(xii) If pc is a wff then so is ' $\forall \alpha \in E$ pc' is a wff

(xiii) If pc is a wff then so is ' $\exists \alpha \in E$ pc' is a wff

(xiv) If pc' is defined within pc, then if pc is a wff then so is pc'

(xv) nothing else is a wff

[end of Definition 4.2]

Definition 4.3 An Environment Of Non-deterministic Actions

An environment E of actions is non-deterministic if $\exists$ an equivalence relation called the non-deterministic relation $\nu = \{(\alpha,\beta) \mid \alpha,\beta \in E$ s.t. pc(α,β) = " 'α *or* β' " is

true}. The pc(α,β) = 'α *or* β' is true if it is not possible to state which one of 'α' *or* 'β' is true during a determination. Actions are self referential in the relation. It follows that:

(i) $\alpha \nu \alpha$ ==> " 'α *or* α' " is true

(ii) $\alpha \nu \beta$ ==> $\beta \nu \alpha$ ==> " 'α *or* β' " is true

(iii) $\alpha \nu \beta$ *and* $\beta \nu \gamma$ ==> $\alpha \nu \gamma$ ==> " 'α *or* γ' " is true

[end of Definition 4.3]

The pc: 'α *or* β' is true implies that it is possible to make the determination that one of the actions α *or* β, can occur. Non-determinism implies that is not possible to say which one of α and β will occur. The relation ν induces an equivalence class of non-deterministic actions containing α and β in E at the time of determination. Note, that the modality of determination yields the truth value of the *'or'* constraint and not necessarily the occurrence of an action.

The equivalence classes $[\alpha]_{\nu 1}$ and $[\alpha]_{\nu 2}$ may be disjoint if the corresponding determination is made at different times, or if the pc "not '$\alpha_{\nu 1}$ *or* $\alpha_{\nu 2}$' " is true. Since the first determination identifies the class of actions which are ν_1–related and the second determination identifies the class of actions which are ν_2–related. Note that in any determination, if α is the only pair, then $[\alpha]_\nu = \{\alpha\}$ and 'α' is true and 'α *or* α' is true. If 'α' is true and 'α *or* α' not true, then E is not non-deterministic. This means that E is a deterministic environment with respect to the determination 'α'.

Example 4.2

In Figure 4.3 the poset chains of process P2 are a representation of a non-deterministic environment, while the α of P1 is in a deterministic E. If we make 'α *or* α' not true, i.e. only 'α' is true initially in P2 then P1 and P2 become observationally

equivalent. Such a behaviour can be defined by superimposing the initial 'α's and the rest of the two chains of P2.

[end of example 4.2]

Definition 4.4. An Environment Of Concurrent Actions

An environment E of actions is concurrent if $\exists$ an ordering relation called the concurrent relation $\chi = \{(\alpha,\beta) \mid \forall\ \alpha,\beta \in E$ s.t. pc(α,β) = " '$\alpha < \beta$' *or* '$\beta < \alpha$' " is true$\}$. The pc(α,β) is true,

if 'α precedes β *or* β precedes α'. It follows that:

(i) $\alpha \chi \beta \Longrightarrow \beta \chi \alpha \Longrightarrow$

" '$\alpha < \beta$' *or* '$\beta < \alpha$' " *or* " '$\beta < \alpha$' *or* '$\alpha < \beta$' "

[end of definition 4.4]

The 'or' is the non-deterministic '*or*' defined in Definition 4.3. The proposition '$\alpha < \beta$' is interpreted to be true if 'α precedes β'. That means that a determination of concurrent action in E results in the assertion:

" '$\alpha < \beta$' *or* '$\beta < \alpha$' " being true. As a consequence,

'$\alpha \chi \beta$' ==> 'α *or* β' is true, but not necessarily vice-versa.

Example 4.3

The actions a and b in (4.1)

[end of example 4.3]

Definition 4.2 and Example 4.3 lead to:

Lemma 4.1

The concurrent relation refines the non-deterministic relation.

Proof:

We must show that $[\alpha]_V \supseteq [\alpha]_\chi$.

Action $\alpha =_\chi \beta$ ==> " '$\alpha < \beta$' *or* '$\beta < \alpha$' "

==> " 'α precedes -an action-' *or* 'β precedes -an action-' "

==> 'α *or* β' is true

==> $\alpha =_V \beta$

==> $[\alpha]_V \supseteq [\alpha]_\chi$

[end of proof]

Example 4.4

If the actions α and β represent input/output actions, then a possible behaviour description of a one place communication buffer can be described by:

$$Buf = \alpha.\beta + \beta.\alpha \qquad (4.12)$$

where if α is an input action then β is an output action and vice versa.

Assume that a satisfaction relation between the process algebra of the specification in Equation 4.12 and partial constraints exists, Equation 4.12 then satisfies the pc: " 'α precedes β' *or* 'β precedes α' ", which implies that it satisfies the pc: 'α *or* β' is true, from Lemma 4.1.

[end of example 4.4]

Definition 4.5 An Environment Of Communication

A non-deterministic environment E is a communication environment, if $\exists$ an equivalence relation called the communication relation $\mu = \{(\alpha,\beta) \mid \alpha,\beta \in E$ s.t. pc(α,β) = " '$\alpha \leq \beta$' *or* '$\beta \leq \alpha$' " is true$\}$.

(i) $\alpha \mu \alpha$ <==> " '$\alpha \leq \alpha$' " is true

(ii) $\alpha \mu \beta$ ==> $\beta \mu \alpha$ <==> " '$\alpha \leq \beta$' *or* '$\beta \leq \alpha$' " is true

(iii) $\alpha \mu \beta$ and $\beta \mu \gamma \Rightarrow \alpha \mu \gamma \Leftrightarrow$ " '$\alpha \leq \gamma$ *or* '$\gamma \leq \alpha$' " is true.

[end of Definition 4.5]

The 'or' is the non-deterministic *'or'* defined in Definition 4.3.2. The proposition '$\alpha \leq \beta$' is interpreted to be true if " 'α precedes β' *or* 'α *and* β' " is true. That means that a determination results in the pc " '$\alpha \leq \beta$' *or* '$\beta \leq \alpha$'" being true. Equality is interpreted as the possible simultaneous occurrence of both α *and* β. As a consequence, 'α *and* β' ==> 'α *or* β' is true, but not necessarily vice-versa.

Example 4.5

In Figure 4.3, If the initial αs of the two chains of P2 occur simultaneously (or synchronise), represented by superimposing the initial α nodes, then 'α *and* α' is true for P2,which implies 'α *or* α' is true.

[end of example 4.5]

Lemma 4.2

The communication relation refines the non-deterministic relation.

The proof is similar to that of Lemma 4.1

[end of Lemma 4.2]

Lemma 4.3

The actions of a communication environment cannot be totally ordered.

Proof:

It is required to show that for a total ordering on E, it should be possible to find a least element. Suppose, α is the least action in E, then we can obtain the single chain:

i.e. $\alpha \leq \beta \leq \gamma \leq \delta$... for all the actions of E

but from Lemma 4.2 E is also non-deterministic ==> $\alpha \vee \alpha$ is true, which implies we can obtain at least two chains with two least elements one with α and another (also) with α. Therefore, E cannot be totally ordered. It is only possible to *partially* order E.

[end of proof]

Example 4.6

The Buf of the last example can also specified as:

$$Buf=\alpha.\beta + \alpha.\alpha \qquad (4.13)$$

[end of example 4.6]

Definition 4.6 A Computational Environment E

A communication environment E is a computational environment, if a function f can be defined on E s.t. $\exists\ \alpha,\beta,\gamma \in E$ s.t.

(i) $\alpha\ \mu\ \beta$ and $f(\alpha,\beta) = \gamma$ <==> "'$\alpha \leq \beta$' *or* '$\beta \leq \alpha$' *or* 'γ'"

[end of Definition 4.6]

Example 4.7

Again for the buffer of the last example, if γ is an action such that

" '$\alpha \leq \gamma \leq \beta$' *or* '$\beta \leq \gamma \leq \alpha$' ", and if the behaviour of the buffer is specified as:

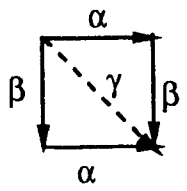
$$Buf = \alpha.\beta + \beta.\alpha + \gamma \qquad (4.14)$$

Then a function f can be defined such that the functionality of 'Buf' is equivalent to any one of the the terms of Equation (4.14). And if each term computes the same functionality then by a satisfaction relation similar to that of Example 4.5, the truth of "'$\alpha \leq \beta$' *or* '$\beta \leq \alpha$' *or* 'γ'" can be derived.

Additionally, if γ is omitted in " '$\alpha \leq \gamma \leq \beta$' *or* '$\beta \leq \gamma \leq \alpha$' ", we obtain the constraint " '$\alpha \leq \beta$' *or* '$\beta \leq \alpha$' ", which implies $\alpha\ \mu\ \beta$.

Also, if $\alpha\ \mu\ \beta$, and $\alpha = \beta$, then "'$\alpha \leq \gamma \leq \beta$' *or* '$\beta \leq \gamma \leq \alpha$'" ==> $\alpha = \beta = \gamma$.

The behaviour of 'Buf' can also be represented as a multigraph in which each node corresponds to a point in the computation and an edge from that node corresponds to an action transition leading to the next point in the computation. Branches in the graph represent non-deterministic choices. The graph for communication behaviour is given below:



[end of example 4.7]

Theorem 4.1

f is well defined but not totally defined.

Proof:

(i) By definition the image of f under μ is defined as:

For $\alpha \mu \beta$, $f(\alpha,\beta) = \gamma$ <==> " '$\alpha \leq \beta$' *or* '$\beta \leq \alpha$' *or* 'γ' "

As in Example 4.7, γ can be chosen such that:

"'$\alpha \leq \gamma \leq \beta$' *or* '$\beta \leq \gamma \leq \alpha$'"

Similarly,

for $\alpha \mu \beta$ and $f(\alpha,\beta) = \gamma^{\wedge}$ <==>

" '$\alpha \leq \beta$' *or* '$\beta \leq \alpha$' *or* '$\gamma^{\wedge}$' "

and $\gamma^{\wedge}$ can be chosen such that:

"'$\alpha \leq \gamma^{\wedge} \leq \beta$' *or* '$\beta \leq \gamma^{\wedge} \leq \alpha$'"

From which

" '$\gamma \leq \gamma^{\wedge}$' *or* '$\gamma^{\wedge} \leq \gamma$' "

Therefore: $\gamma = \gamma^{\wedge}$.

(ii) f is not totally defined.

Take $\alpha,\beta \in E$ s.t. $\alpha\ \mu$ (not μ) β but $\alpha\ \nu\ \beta$, $f(\alpha,\beta)$ is undefined and therefore not total.

[end of proof]

The following is another example in which two concurrent actions do not have the same functionality.

Example 4.8

Suppose α corresponds to an input action only and β an output action only then the behaviour of buffer can be represented as:

$$Buf = \alpha.\beta \neq \beta.\alpha \qquad (4.15)$$

The terms $\alpha.\beta$ and $\beta.\alpha$ do not have the same functionality.

[end of example 4.8]

Indeed, communication is intrinsic to the non-deterministic occurrence of events in a partially ordered behaviour.

Theorem 4.2

The intersection of the communication relation μ and a partial order relation π (obtained from the DT of communicating actions) defines a computation in E. See Figure 4.4

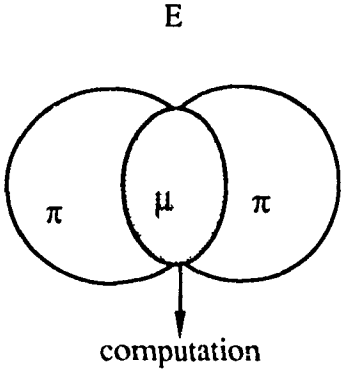


Figure 4.4 A computation in a communication environment

Proof:

Suppose, $\alpha,\beta \in \mu$ ==> " '$\alpha \leq \beta$' *or* '$\beta \leq \alpha$' " is true, and $\alpha \leq_\pi \beta$ *or* $\beta \leq_\pi \alpha$.

Since, $\alpha,\beta \in \mu$ ==> $\alpha,\beta \in \nu$ ==> 'α *or* β' is true ==>

$\exists$ an action γ s.t. a partial order π on E can be defined given by:

$\alpha \leq \gamma \leq \beta \leq \gamma \leq \alpha \leq ...$

or

$\beta \leq \gamma \leq \alpha \leq \gamma \leq \beta \leq ...$

where " '$\alpha \leq \beta$' *or* '$\beta \leq \alpha$' " is true, and $\alpha \leq_\pi \beta$ *or* $\beta \leq_\pi \alpha$ is true.

We can now choose a partial function f s.t. $f(\alpha,\beta) = \gamma$. Hence, $\mu \cap \pi$ defines a computation.

[end of proof]

Corollary 1.

For a finite number of communicating actions in E, intersections of partial orders on E define computations of partial communication behaviour.

Proof:

Consider two partial orders on the actions of E:

$\alpha 1 \leq \beta 1 \leq \gamma 1 \leq \delta 1...$ and

$\alpha 2 \leq \beta 2 \leq \gamma 2 \leq \delta 2...$ and

suppose that $\alpha 1\ \mu\ \alpha 2$, $\beta 1\ \mu\ \beta 2$,... then a partial order can be defined s.t.:

$\alpha 1 \leq \alpha 2 \leq \beta 1 \leq \beta 2 \leq \gamma 1 \leq \gamma 2 \leq \delta 1 \leq \delta 2....$

or (4.16)

$\alpha 2 \leq \alpha 1 \leq \beta 2 \leq \beta 1 \leq \gamma 2 \leq \gamma 1 \leq \delta 2 \leq \delta 1....$

Taking intersections $\mu \cap \pi$ of equal, finite lengths of Equation (4.16) results in a computation of partial communication behaviour: So that (4.16) can be written as:

$\alpha 1 = \alpha 2 \leq \beta 1 = \beta 2 \leq \gamma\ 1 = \gamma 2$ ...

Replacing $\alpha 1 = \alpha 2$ by α, $\beta 1 = \beta 2$ by β and so on we get:

$\alpha \leq \beta \leq \gamma \leq \delta....$

a partially ordered computation.

[end of proof]

The theorem shows that a computation can be defined as the intersection of partial orders on E and a communication relation μ.

Hence, if we take intersections of a finite number of partial orders and communication relations on actions, a finite number of computations result, thus providing a partial order structure to a communication behaviour in E. Therefore, if a process algebra is selected for mapping this behaviour, a process in that algebra represents the structure of such a computation.

Corollary 2.

The computational structure of the communication behaviour of a finite number of actions can be represented by an infinite recursion of processes.

Proof:

For a finite number of actions in E, consider the infinite partial orders:

$$\alpha 1 \leq \alpha 2 \leq \beta 1 \leq \beta 2 \leq \gamma 1 \leq \gamma 2 \leq \delta 1 \leq \delta 2 \ldots.$$

or (4.17)

$$\alpha 2 \leq \alpha 1 \leq \beta 2 \leq \beta 1 \leq \gamma 2 \leq \gamma 1 \leq \delta 2 \leq \delta 1 \ldots.$$

To define a process, we assume that the set of actions $\alpha 1$, $\alpha 2, \beta 1, \beta 2, \gamma 1, \gamma 2, \delta 1, \delta 2 \ldots$. is finite. This implies that in an infinite partial order of the type in Equation (4.17) the finite number of labels begin to recur. From which we deduce that there exists some finite length sequence in an infinite partial order beyond which the partial order is periodic modulo that sequence length. That means that every finite portion of the partial orders is repeated infinitely often.

But by the theorem, finite parts of the infinite partial orders can represent a finite computation. From which it follows that communication behaviour can be represented by infinitely recursing processes.

[end of proof]

We illustrate by means of the simple one place communication buffer.

Example 4.9

(i) Assume that the buffer of Example 4.7 outputs whatever is input, so that if α,β are input/output actions and $\alpha \mu \beta$, with a function $f(\alpha,\beta) = \gamma$, then every intersection of the communication relation and the partial orders:

$$\alpha \leq \gamma \leq \beta \leq \gamma \leq \alpha \leq \gamma \leq \beta...$$

or (4.18)

$$\beta \leq \gamma \leq \alpha \leq \gamma \leq \beta \leq \gamma \leq \alpha \leq \gamma...$$

gives the constraint: " '$\alpha \leq \gamma \leq \beta$' *or* '$\beta \leq \gamma \leq \alpha$' " and the process algebraic behaviour:

$$Buf = \alpha.\beta + \beta.\alpha + \gamma \quad (4.19)$$

This behaviour is finite and therefore terminates.

The recursive behaviour:

$$Buf = (\alpha.\beta + \beta.\alpha + \gamma).Buf \quad (4.20)$$

of the buffer process, also, always satisfies the constraints, although it is non-terminating behaviour.

The functionality of the buffer process is given by any one of the terms of Equation (4.20), i.e. the action γ or the cross-product of the functionalities of the actions α,β. Therefore, omitting γ does not reduce the functionality of the specification, so that the partial order can be written as:

$$\alpha \leq \beta \leq \alpha \leq \beta....$$

or (4.21)

$$\beta \leq \alpha \leq \beta \leq \alpha \leq....$$

Intuitively, the omission of γ in the behaviour specification is related to the 'depth' of the determination. The depth of a determination can be measured by the depth of the associated constraint.

[end of example 4.9]

Formally:

Definition 4.8 Depth Of A Determination

The depth of a determination is measured by the depth of the associated partial constraint on a given set of actions. The depth of the associated partial constraint is the number of atomic actions bound by the precedence operator, or the depth of determination is 1.

[end of Definition 4.8]

Example 4.10

If a determination on the actions α,β in E results in the assertion that

(i) 'α *or* β' is true, the depth of the determination = 1

(ii) 'α precedes α' *or* 'β precedes β' is true, the depth of the determination = 2.

(iii) 'α precedes β *or* β precedes α' is true, the depth of the determination = 2.

(iv) 'α precedes β precedes γ' is true, the depth of the determination = 3.

Note that the first two terms in (4.19) and Equation (4.20) are the result of a determination of depth two, while the last one is the result of a determination of depth one. That means the behaviour of the Buf at depth one is: $\alpha + \beta + \gamma$. So that the action γ appears at both depths. Consistency between the determinations depths 1 and 2 is achieved by choosing γ to be an internal, unobservable action which is equal to both α *and* β. This is further explained later.

[end of example 4.10]

Thus, partial constraints on system behaviour, considered from the point of view of depths of determination, represent the limits within which distributed system behaviour can be specified.

Theorem 4.3

Non-determinism and communication represent a minimum and a maximum constraint (boundary) respectively on the partial definition of communication behaviour.

Proof:

1. For actions α,β: $\alpha\mu\beta$ and $f(\alpha,\beta) = \gamma$, we can choose γ s.t.:

" '$\alpha \leq_\pi \gamma \leq_\pi \beta$' *or* '$\beta \leq_\pi \gamma \leq_\pi \alpha$' "

==> " '$\alpha \leq_\pi \beta$' *or* '$\beta \leq_\pi \alpha$' "

==> " '$\alpha < \beta$' *or* '$\beta < \alpha$' *or* '$\alpha = \beta$' "

Where the omission of γ implies that the functionality is the product of the functionalities of the actions α *and* β, or the functionality of α, or of β.

If '$\alpha < \beta$' *or* '$\beta < \alpha$' is true, the functionality is that of the product of α *and* β - a maximum for the behaviour.

If '$\alpha = \beta$' is true, the functionality is that of of α, or of β - a minimum.

[end of proof]

This result can be interpreted in terms of action trees as follows. A determination associated with action tree growth starting from the root node, will result in determination of depth one corresponding to the tree given by '$\alpha + \beta$', and a determination at depth two is given by " 'α precedes β' *or* 'β precedes α' *or* '$\alpha = \beta$' *or* 'γ'". The corresponding trees are shown in Figure 4.5.

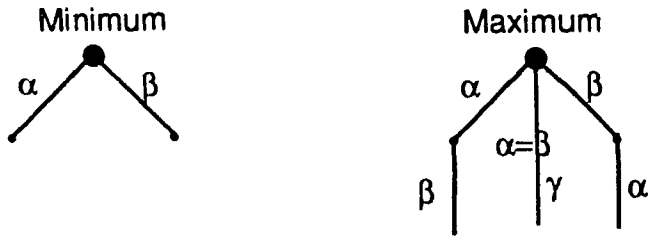


Figure 4.5. Minimum and maximum action trees at depth 1 and 2

The architectural implication of a determination at depth=2 means that, if a determination is made on a system globally, components of the system may exhibit the behaviours: $\alpha\alpha$, $\beta\beta$, $\alpha\beta$,$\beta\alpha$,$\alpha\beta + \beta\alpha$, ..., at depth of determination=2

Example 4.11

Consider now the example of a one place queue which can be used to input an element of data, transmit it via the queue, and output that element.

Let α denote the input action, γ the transmission and β the output action.

The behaviour of the queue satisfies the pc:

" 'α precedes γ precedes β' " ;

a total order corresponding to this constraint can be described as:

$$\alpha < \gamma < \beta < \alpha < \gamma < \beta < \alpha < \gamma < \beta ... \quad (4.22)$$

and its process algebraic behaviour can be described as:

$$Que = \alpha.\gamma.\beta.Que \quad (4.23)$$

Figure 4.6 represents the queue process as:

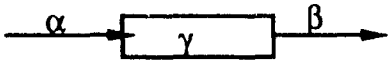


Figure 4.6. The queue process

Suppose now, that we are able to make determinations of input actions and output actions separately (in a distributed manner), then the behaviour of the queue satisfies the constraint:

" 'α precedes γ' and 'γ precedes β' " ;

with the partial orders:

$\alpha<\gamma<\alpha<\gamma<\alpha<\gamma<\alpha...$

and (4.24)

$\gamma<\beta<\gamma<\beta<\gamma<\beta<\gamma...$

respectively.

The behaviours:

$Inque = \alpha.\gamma.Inque$

and (4.25)

$Outque = \gamma.\beta.Outque$

satisfy the constraints, where the queue is distributed between two processes: the Inque process, whereby a user-queue interaction (at the queue interface) leads to determinations implying that input precedes transmission, and the Outque process, which leads to determinations implying that transmission precedes output.

It is not readily apparent how the two processes of Equation (4.25) can be composed together to obtain the behaviour of the queue process in Equation (4.23). What should the operational semantics of the composition operator be? In other words:

Inque "operator" Outque = Que (4.26)

where "operator" is some operator which results in 'Que' on the RHS of Equation 4.26.

One approach may be to consider only finite behaviours of each process (see Figure 4.7) by defining:

$$Que = \alpha.\gamma.\beta \tag{4.27}$$

and:

$$Inque = \alpha.\gamma$$

and

$$Outque = \gamma.\beta \tag{4.28}$$

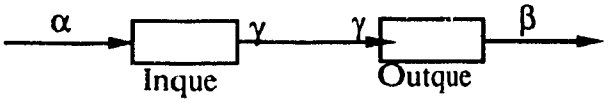
For this finite case, a solution is:

$$\begin{aligned} Que &= Inque.Outque \\ &= \alpha.\gamma.\gamma.\beta \\ &= \alpha.\gamma.\beta \end{aligned} \tag{4.29}$$

Obtained by reasoning that $\gamma.\gamma$ gives γ again, since γ is the same action. But Equation 4.29 is obviously an awkward solution. Another approach would be to parallel compose the two, where the operator ('||') composes common actions and interleaves others:

$$\begin{aligned} Que &= Inque||Outque \\ &= \alpha.\gamma||\gamma.\beta \\ &= \alpha.\gamma.\beta \end{aligned} \tag{4.30}$$

Obviously this expression satisfies the service required from the queue.



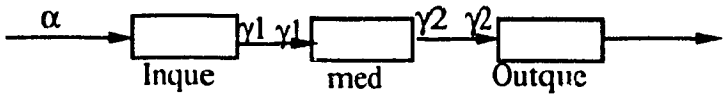


Figure 4.7. Decomposition of queues

Suppose now that we decompose the queue and distribute the transmission action γ as follows (see the second part of Figure 4.7):

Let $\gamma 1$ denote the send action and $\gamma 2$ denote the receive action, or the beginning and the end of transmission, s.t. $\gamma 1.\gamma 2$ (i.e. send precedes receive) or $\gamma 1=\gamma 2=\gamma$.

Now the queue process can be defined as:

$$que = Inque + Outque \tag{4.31}$$

where

$Inque = \alpha.\gamma 1.que$

and

$Outque = \gamma 2.\beta.que$

and

$med = \gamma 1.\gamma 2.med$

and

$usr = usr1 + usr2$

$usr1 = \alpha.usr$

and

$usr2 = \beta.usr$

and define a closed form of parallel composition where only common actions can occur:

$$Que = quell(med + usr) \tag{4.32}$$

or

$Que = (Inque + Outque) \,||\, (usr1 + med + usr2)$

$= (\alpha.\gamma 1.que + \gamma 2.\beta.que) \,||\, (\alpha.usr + \gamma 1.\gamma 2.med + \beta.usr)$

$= (.\gamma 1.que + \gamma 2.\beta.que) \,||\, (.usr + \gamma 1.\gamma 2.med + \beta.usr)$

(α is the interaction)

$$= (..que + \gamma2.\beta.que) \parallel (.usr + .\gamma2.med + \beta.usr)$$

($\gamma1$ is the interaction)

$$= (..que + .\beta.que) \parallel (.usr + ..med + \beta.usr)$$

($\gamma2$ is the interaction)

$$= (..que + ..que) \parallel (.usr + med + .usr)$$

(β is the interaction)

(4.33)

Common actions are in a communication relation and their occurrence corresponds to communication behaviour. Equation 4.33 replaces actions which have already occurred by dots. Traces of interactions show that recursion occurs with the actions:

$\alpha,\gamma1,\gamma2,\beta$

or

α,γ,β (if $\gamma1=\gamma2=\gamma$) (4.34)

If γ is structurally hidden in this computation, the partial order of actions in Equation (4.34) can be written as:

$\alpha<\beta<\alpha<\beta<\alpha<\beta...$ (4.35)

which satisfies the pc: " '$\alpha<\beta$' *or* γ ".

Alternatively, if the entire computational structure is hidden, i.e. the actions input and output are somehow now internal actions than it suffices to specify the whole structure by the internal action action γ, and the partial order can be represented by:

$...\gamma\leq\gamma\leq\gamma\leq\gamma\leq\gamma\leq\gamma\leq\gamma....$

or (4.36)

$...\gamma=\gamma=\gamma=\gamma=\gamma=\gamma=\gamma....$

Note that the same result can be obtained from the following parallel composition:

usr1||Inque||med1||med2||Outque||usr2 (4.37)

[end of example 4.11]

Internalisation or hiding of some events in a distributed system implies that consecutive determinations would obtain observable actions or hidden actions. Observable actions of a distributed system are, in general, non deterministic actions, while internal actions are related to communication interactions with an underlying service. This fact suggest the statement of the following:

Theorem 4.4

Communication behaviour can be computed partially such that it satisfies non-determinism and communication constraints in alternation.

Proof:

Suppose $\alpha \mu \beta$ and $\exists$ an f and γ s.t $f(\alpha,\beta) = \gamma$ and,

==> " '$\alpha \leq_\pi \gamma \leq_\pi \beta$' or '$\beta \leq_\pi \gamma \leq_\pi \alpha$' " and $\alpha\mu\beta$

==> " '$\alpha \leq_\pi \beta$' *or* '$\beta \leq_\pi \alpha$' " and $\alpha\mu\beta$

where 'γ' is omitted from the partial order relation between α and β as in Theorem 4.2.

Therefore:

" 'α *or* β' or '$\alpha < \beta$ *or* $\beta < \alpha$' or '$\alpha = \beta$' *or* 'γ' " is true.

Therefore, the computational structure of f can be represented such that consecutive determinations alternate between components of the computational structure. That means determinations derive the pc 'α *or* β' or '$\alpha < \beta$ *or* $\beta < \alpha$' *or* 'γ' depending on which component of the structure is involved in the determination.

[end of proof]

" The truth has no distinctions; These come from our foolish clinging to This and That "
- Hokleyo

Chapter 5

OPEN DISTRIBUTED ARCHITECTURE-A LOGICAL VIEW

In this chapter, the specification of distributed systems using partial constraints is illustrated by means of realistic protocols. The design strategy provided by the partial constraint method relates computational structures of system behaviour to system architecture. Process algebraic specifications resulting from partial constraints are partial with respect to the set of requirements. Validation and verification analysis can be carried on these specifications even though they are partial specifications of system behaviour.

5.1 Constraints And Architecture

Design and specification strategies manage complexity of system behaviour. In chapter 4, example 4.1.1 showed how constraint-oriented specifications in LOTOS decompose complex temporal ordering relations on events of system behaviour into simpler relations. Simpler relations are then composed into compound expressions.

Compound expressions of temporal ordering relations on events are assumed to satisfy given requirement constraints [81]. However, such an assumption presumes a methodological demonstration of satisfaction. Obviously, replacement of the presumption by an explicit definition of the satisfaction relation would aid the design and analysis of system behaviour.

Such a satisfaction relation is given for CCS in [54]. A set of modal formulae are defined on system actions, which can be derived from the transition relations on behaviour expressions of CCS. Therefore, if non-determinism can be included in such formulae, then a definition of derivability of assertions from behaviours would provide the means to define a satisfaction relation which include non-determinism in behaviour expressions.

In Chapter 4, a set of relations between the actions of an environment E were defined to construct partial constraints satisfied by behaviours in E. In this chapter, a satisfaction relation is defined on behaviours of E, expressed as expressions in a generic process algebra, as well as in terms of partial constraints on the actions of E.

Intuitively, constraints constrain communication behaviour. In the engineering sense, constraints on design are boundary values of system functionalities. In this chapter, constraints are viewed as logical limits of specified communication behaviour. That means given constraints on system behaviour, design should result in a specification of behaviour, within the margins established by those constraints. The notion of margins of behaviour and its consequences on system architecture is further examined in the sequel.

But what are the implications of constraints on system architecture? Examination of the architectural implications of constraints is begun by means of the following simple example.

Example 5.1

Suppose, we are required to design a system which communicates messages between users. One user can insert a message into the system which can be subsequently extracted by another user.

Let action α be the result of user-system interactions in an environment E. The action α is associated with both operations: (user)insertion-(system)insertion and (user)extraction-(system)extraction.

Assume that the pc: 'α' is derivable from a determination of behaviour in E. This implies that the behaviour of the system satisfies:

Partial Constraint

'α' (5.1)

'α' is true implies that either the operation 'insertion' or the operation 'extraction' (of a message) is always true. To keep the example simple we do not describe the 'message' parameter of the operations.

The Partial Constraint (5.1), constrains the behaviour of the system under design. Choosing the CCS process model for example, specification of system behaviour involves the mapping of constraints onto a CCS behaviour specification. The behaviour so specified in turn, must satisfy this constraint.

Satisfaction of the constraint by the behaviour S implies that, S must always satisfy the Partial Constraint (5.1).

If the messaging system is represented by the CCS process S, the behaviour satisfying the Partial Constraint 5.1 can be expressed as:

$S = \alpha.S$ (5.2)

S is intended to satisfy the required communication service:

(i) users may insert a message into the system (action α)

(ii) users may extract a message from the system (action α).

But, the specification S does not distinguish between a user inserting a message, and a user extracting a message. More importantly, if S represents the communication service obtained from a distributed system, then S is in fact a composition of distributed processes, S1 and S2 (say).

If the system S is distributed into processes S1 and S2, then the communication service implied by the Partial Constraint 5.1 is that users may use process S1 or S2 for insertion/extraction of messages via the action α. And the specification:

$$\begin{aligned} S &= S1 + S2 \\ &= \alpha.S + \alpha.S \\ &= (\alpha + \alpha).S \\ &= \alpha.S \end{aligned} \qquad (5.3)$$

where $S1 = \alpha.S$ *and* $S2 = \alpha.S$, and $S = S1+S2$, describes the intended service. But since S is distributed, users using process S1 cannot use S2 and vice versa. The architectural requirement which must be satisfied by the behaviour description is that user interactions with S1 must be distinguishable from those with S2. And a determination on the behaviour of S must express behaviour of S1 or S2 - non-deterministically. The distinguishing of usage of S1 and S2 can be accomplished by distinguishing between the actions of S1 and S2.

Therefore, we let the messaging system be represented by process S, and the actions α and β denote both operations, insertion and extraction. Accordingly, a determination of the behaviour of S must satisfy:

Partial Constraint

'α *or* β' (5.4)

The behaviour equation:

$$S = (\alpha + \beta).S \tag{5.5}$$

and the distributed expression:

$$S = S1 + S2$$

$$= \alpha.S + \beta.S \tag{5.6}$$

satisfy the Partial Constraint (5.4). Where $S1 = \alpha.S$ and $S2 = \beta.S$. The corresponding action tree for the constraint is shown in Figure 5.1:

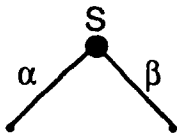


Figure 5.1. An 'or' constraint as a behaviour action tree

The Partial Constraint (5.4) is a logical assertion which describes requirements on system behaviour, without implying any system architecture. Equations 5.5 and 5.6 however specify CCS processes and hence implicitly suggest system architectures.

Figures 5.2 and 5.3 show the non-distributed and distributed processes, S, S1 and S2 respectively. In the figures, a process is represented as a box and the actions α and β by the double-headed arrows. The actions α and β occur at user-process interfaces. Users are considered to be part of the environment and not explicitly represented. The line connecting S1 and S2 in Figure 5.3 indicates the possible insertion (say), of a message via action α with S1 and extraction (say) of that message via action β with S2.

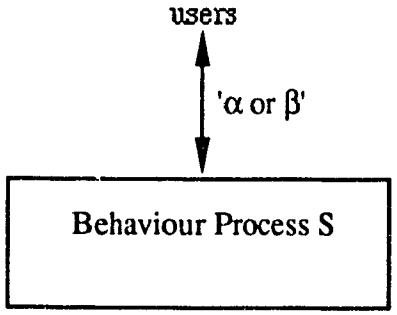


Figure 5.2. Process architecture of system behaviour

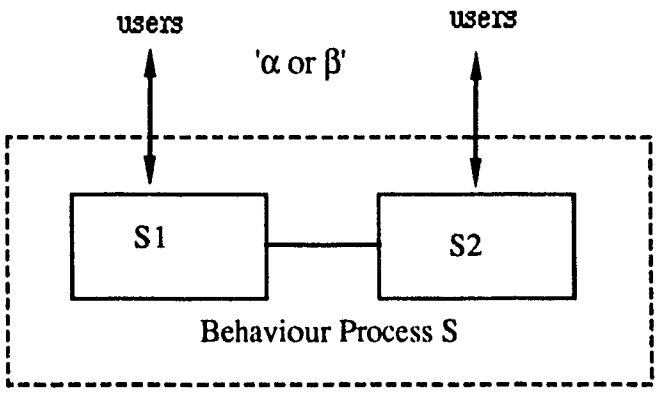


Figure 5.3. Distributed Architecture of Process S

In this way, architecture can be viewed as the spatial projection (implicit or explicit) of logical constraints. For example, the architectural requirements of the Partial Constraint are satisfied by the behaviour of the process specified in Equation 5.5. The process in Figure 5.2 can be interpreted as an architectural model of S. Similarly, the architectural requirements of the Partial Constraint 5.4 are also satisfied by the processes of the behaviour in Equation 5.6 represented in Figure 5.3.

Suppose now, that a determination of depth 2 is made on the behaviour of S, whereby an insertion via action α *or* β is followed by an extraction or insertion via action β *or* α. This behaviour satisfies the:

Partial Constraint

'"α precedes β' *or* 'β precedes α'" (5.7)

Consider now, only part of the Partial Constraint (5.7):

Partial Constraint

'α precedes β' (5.8)

which can be expressed as:

$$S = \alpha.\beta.S \tag{5.9}$$

for the monolithic system S. And its action tree as:

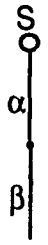


Figure 5.4. A precedence constraint as a behaviour action tree

[end of example 5.1]

The truth of the precedence constraint at depth of determination 2 implies that a β (associated with an extraction operation) action must be preceded by an α (associated with an insertion) action in order that the system provide the required communication service to users.

Intuitively, the 'or' constraint on two actions, may be interpreted as being satisfied by the behaviour's action tree (Figure 5.1) upto depth one (what the system *may* do). While the precedence constraint (Figure 5.4) is satisfied by the behaviour's action tree upto depth two (what the system *must* do).

In terms of limits on system behaviour, what the system *may* do represents a minimum constraint or a lower bound, and what the system *must* do corresponds to a maximum constraint, or an upper bound.

5.2 Minimum And Maximum Constraints

Comparing the non-deterministic constraints of Figure 5.1 and the precedence constraints of Figure 5.4 with those of Definitions 4.4.3, 4.4.5 and Theorem 4.4.3, it can

be seen that they represent the minimum and maximum functionalities of system behaviour.

Indeed, the architectural constraints are just the minimum and maximum (Theorem 4.4.3) of the communication constraint. The minimum and maximum functionalities must be satisfied by the corresponding computational structure of specified behaviour, and hence by its architecture. This means, whereas a monolithic computational structure may be used to satisfy the constraint with maximum functionality only, a distributed computational structure may allow some of its components to satisfy a minimum functionality of the system.

But how do we obtain the maximum functionality of the system from distributed components with minimum functionality?

5.2.1 Uni-directional Communication

In case system architecture is distributed with processes S1 and S2, a determination of depth 2 for the same service as that satisfying the Partial Constraint (5.8) necessitates the definition of a dummy action γ corresponding to the transmission of the message from S1 to S2 (see Example 4.11). Consider the following example:

Example 5.2

Partial Constraint

('α precedes γ *or* 'γ precedes β') " (5.10)

must be satisfied by the behaviours S1 and S2.

But the action γ has to be distinguished when associated with either of the processes, S1 or S2. Therefore, let $\gamma=\gamma1=\gamma2$ and the Partial Constraint 5.10 can be written as the Partial Constraint (5.11):

('α precedes $\gamma1$' *or* '$\gamma2$ precedes β') " (5.11)

Accordingly, the processes S1 and S2 can be defined as:

$S1 = \alpha.\gamma1.S$

and

$S2 = \gamma2.\beta.S$; where $S = S1 + S2$ (5.12)

Alternatively, S1 and S2 can also be defined as:

$S1 = \alpha.\gamma1.S1$

and

$S2 = \gamma2.\beta.S2$; (5.13)

The difference between the two forms of distribution is examined in the sequel.

Distribution of the architecture of S implies its decomposition into two sub-processes. But the two sub-processes must communicate in order to provide the intended service. Therefore, distribution also imposes another, seemingly contradictory, architectural constraint - that system architecture must contain a non-distributed component to enable the distributed sub-processes to communicate. That means the architecture must contain a single undivided process - the underlying service to be used for communication. The transmission action γ is the result of a determination on the behaviour of such a single undivided process, the 'med' (say) s.t. γ is the result of communication between two actions $\gamma1$ and $\gamma2$. The process 'med' must satisfy the constraint:

'$\gamma1$ precedes $\gamma2$' (5.14)

and the definition:

$$med = \gamma 1.\gamma 2.med \qquad (5.15)$$

This seeming contradiction of constraints is a basic characteristic of the specification of communication behaviour, but is reasonable.

The reasonableness of such a constraint becomes evident if it can be shown that Equation (5.9) can be obtained from Equation (5.12) and Equation (5.15), for this will prove that the distributed system satisfying contradictory constraints provides the same service as the monolithic one.

Consider first, the margin of behaviour between these two limits? Since the upper limit expresses the intension of the computation and the lower limit the extension of computation, there exists a behaviour which relates the two behaviours. That behaviour is the realisation of communication between the distributed and non-distributed components of the system and the computation of the functionality of the system.

The pc which must be satisfied by the system can be expressed as the:

Partial Constraint

('α precedes $\gamma 1$' *or* '$\gamma 2$ precedes β')

and

'$\gamma 1$ precedes $\gamma 2$' (5.16)

The Partial Constraint (5.16) must be satisfied by a behaviour expression which computes the functionality of the distributed system such that the functionality of the monolithic system and its intended service is obtained. Mapping the *'and'* of the constraint to the parallel operator (|) of CCS gives the expression:

$$(S1 + S2) \mid med \qquad (5.17)$$

This CCS operator interleaves actions which are not common to the two components of the parallel operator and synchronises common actions, selecting an action from either side recursively. Interleaved operations are assumed to synchronise or communicate with common actions in the environment of the system S. Therefore:

$$
\begin{aligned}
&(S1 + S2) \mid med \\
&\quad = (\alpha.\gamma1.S + \gamma2.\beta.S) \mid (\gamma1.\gamma2.med) \\
&\quad = (.\gamma1.S + \gamma2.\beta.S) \mid (\gamma1.\gamma2.med) && \textbf{(α is interleaved)} \\
&\quad = (..S + \gamma2.\beta.S) \mid (.\gamma2.med) && \textbf{($\gamma1$ is a communication)} \\
&\quad = (..S + .\beta.S2) \mid (.med) && \textbf{($\gamma2$ is a communication)} \\
&\quad = (..S + ..S) \mid (.med) && \textbf{(β is interleaved)}
\end{aligned}
\qquad (5.18)
$$

where once an interaction involves a selected action of a term in an expression, it is equivalent to a commitment that the behaviour will follow that path till it can again recurse to the original expression. The synchronisation of actions in the two paths is shown by replacing the actions on the two sides of the parallel expression with dots.

Therefore:

$$
\begin{aligned}
&(S1 + S2) \mid med \mid= \\
&\qquad (\alpha,\gamma1,\gamma2,\beta) && \text{(a trace expression)} \\
&\qquad ==> (\alpha.\gamma1.\gamma2.\beta) && \text{(a behaviour expression)} \\
&\qquad ==> (\alpha.\gamma.\gamma.\beta) && \text{(since } \gamma1=\gamma2=\gamma) \\
&\qquad ==> (\alpha.\beta) && \text{when } \gamma \text{ is structurally hidden} \\
&\qquad ==> (\alpha.\beta) && \text{recursively}
\end{aligned}
\qquad (5.19)
$$

Equation (5.19) shows that the behaviour ($\alpha.\beta$) can be recursively obtained for the system S.

The partial behaviour (of S):

$$S = (\alpha.\beta).S \qquad (5.20)$$

satisfies the Partial Constraint (5.8).

[end of example 5.2]

With the alternative definition of S1 and S2 as given in Equation (5.13) the intended service given in Equation (5.20) can also be obtained from the expression:

$$S1|med|S2 \qquad (5.21)$$

The difference between the two is in the efficiency of simulation. The Equation (5.17) provides for a lesser search time for synchronisable actions than Equation (5.21). But the latter provides a cleaner distribution of the sub-processes S1 and S2.

5.2.2 Bi-directional Communication

But a distributed S, consists of the processes S1,S2 and 'med', and the environment of S also contains distributed users of S1 and S2, which must be able to insert and extract messages. This means both S1 and S2 should be able to perform a α and a β action. So that the constraint on the behaviour of S can be written as in the following example:

Example 5.3

Partial Constraint

"'α precedes β' *or* 'β precedes α'" (5.22)

The Partial Constraint (5.22) is the result of a determination of depth two on the system S, whereas the Partial Constraint (5.4) is the result of a determination of depth 1.

Corresponding action trees for both the partial constraints (5.4) and (5.22) are shown in Figure 5.5.

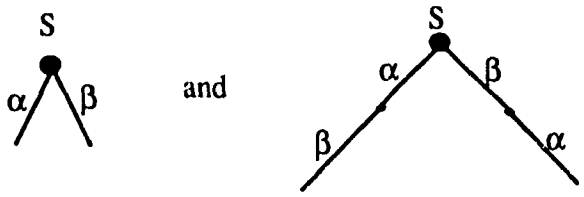


Figure 5.5. Choice and precedence trees of a two-way communication

The behaviour of the system satisfying the Partial Constraint (5.22) is given by:

$$S = (\alpha.\beta + \beta.\alpha).S \qquad (5.23)$$

In a manner similar to the above, we can now consider a distributed architecture and define the processes S1,S2 and the underlying service (the medium) as the:

Partial Constraint

"'γ1 precedes γ2' *or* 'γ2 precedes γ1'" (5.24)

The two partial constraints jointly constrain the system such that they can be expressed as the:

Partial Constraint

"'α precedes γ1' *or* 'γ1 precedes α'

or

'β precedes γ2' *or* 'γ2 precedes β' "

and

"'γ1 precedes γ2' *or* 'γ2 precedes γ1'" (5.25)

The behaviour expressions in the distributed architecture which satisfy the different parts of this constraint are given by:

$$S1 = (\alpha.\gamma1 + \gamma1.\alpha).S ;$$

$$S2 = (\beta.\gamma2 + \gamma2.\beta).S ;$$

$$med = (\gamma1.\gamma2 + \gamma2.\gamma1).med \qquad (5.26)$$

and the behaviour satisfying the partial constraint (5.24) is given by:

$$S1 + S2 \mid med$$

$$\mid= (\alpha.\gamma1 + \gamma1.\alpha).S + (\beta.\gamma2 + \gamma2.\beta).S \parallel (\gamma1.\gamma2 + \gamma2.\gamma1).med$$

$$= (.\gamma1 + \gamma1.\alpha)S + (\beta.\gamma2 + \gamma2.\beta).S \parallel (.\gamma2 + \gamma2.\gamma1).med$$

(α is a possible interleaving action)

$$= (.. + \gamma1.\alpha).S + (\beta.\gamma2 + \gamma2.\beta).S \parallel (.\gamma2 + \gamma2.\gamma1).med$$

($\gamma1$ is a possible interaction)

$$= (.. + \gamma1.\alpha).S + (\beta.\gamma2 + .\beta).S \parallel (.. + \gamma2.\gamma1).med$$

($\gamma2$ is a possible interaction)

$$= (.. + \gamma1.\alpha)S + (\beta.\gamma2 + ..).S \parallel (.. + \gamma2.\gamma1).med$$

(β is a possible interaction)

(5.27)

So that a possible behaviour of the system is derived from the parallel composition expression:

$(S1 + S2) \mid med \mid=$

$(\alpha.\gamma1.\gamma2.\beta)$	(behaviour expression)
$==> (\alpha.\gamma.\gamma.\beta)$	since $\gamma1=\gamma2=\gamma$
$==> (\alpha.\beta)$	If γ is structurally hidden
$==> (\alpha.\beta).S$	is a partial behaviour of S

(5.28)

Similarly:

(S1 + S2) | med |=

	$(\beta.\gamma2.\gamma1.\alpha)$	(behaviour expression)
	==> $(\beta.\gamma.\gamma.\alpha)$	since $\gamma1=\gamma2=\gamma$
	==> $(\beta.\gamma.\alpha)$	
	==> $(\beta.\alpha)$	If γ is structurally hidden
...	==> $(\beta.\alpha).S$	is the recursive behaviour of S

(5.29)

Evidently other possible behaviours can also be derived from this parallel composition:

S1 + S2 | med

|= $(\alpha.\gamma1 + \gamma1.\alpha).S + (\beta.\gamma2 + \gamma2.\beta).S \mid (\gamma1.\gamma2 + \gamma2.\gamma1).med$

= $(\alpha.\gamma1 + .\alpha).S + (\beta.\gamma2 + \gamma2.\beta).S \parallel (.\gamma2 + \gamma2.\gamma1).med$

($\gamma1$ is a possible interaction)

= $(\alpha.\gamma1 + ..)S1 + (\beta.\gamma2 + \gamma2.\beta).S2 \mid (.\gamma2 + \gamma2.\gamma1).med$

(α is a possible interleaving action)

= $(\alpha.\gamma1 + ..)S + (\beta.\gamma2 + .\beta).S2 \mid (.. + \gamma2.\gamma1).med$

($\gamma2$ is a possible interaction)

= $(\alpha.\gamma1 + ..).S + (\beta.\gamma2 + ..).S \mid (.. + \gamma2.\gamma1).med$

(β is a possible interleaving action)

(5.30)

From which the partial behaviour of the system S: $(\gamma 1.\alpha.\gamma 2.\beta).S$ can also be derived.

Such behaviours do not satisfy the Partial Constraint (5.16). This implies the component partial constraints of the Partial Constraint (5.22) from which the distributed architecture of S was obtained is not uniquely distributed.

However, suppose we consider the medium process to be also distributed into two processes such that it satisfies the constraint:

"'$\gamma 1$ precedes $\gamma 2$' *or* '$\kappa 2$ precedes $\kappa 1$'". (5.31)

This clearly distinguishes the side: S1 or S2 which initiates interactions. The composed partial constraint describing behaviour constraints on the system can be expressed as:

"'α precedes $\gamma 1$' *or* '$\kappa 1$ precedes α'

or

'β precedes $\kappa 2$' *or* '$\gamma 2$ precedes β' "

and

"'$\gamma 1$ precedes $\gamma 2$' *or* '$\kappa 2$ precedes $\kappa 1$'" (5.32)

The distributed architecture of S gives the sub-processes S1,S2 and med1 and med2, where:

$S1 = (\alpha.\gamma 1 + \kappa 1.\alpha).S$;

$S2 = (\beta.\kappa 2 + \gamma 2.\beta).S$;

and

med = med1 + med2

where

$med1 = \gamma 1.\gamma 2.med$;

$med2 = \kappa 2.\kappa 1.med$ (5.33)

And the parallel composed behaviour is given by:

$(S1 + S2) \mid (med1 + med2) \mid =$

$(\alpha.\gamma 1 + \kappa 1.\alpha).S + (\beta.\kappa 2 + \gamma 2.\beta).S$

$\mid$

$(\gamma 1.\gamma 2.med + \kappa 2.\kappa 1.med)$

$=$

$(\alpha.\gamma 1 + \kappa 1.\alpha).S + (.\kappa 2 + \gamma 2.\beta).S$

$\mid$

$(\gamma 1.\gamma 2.med1 + \kappa 2.\kappa 1.med2)$

(β is a possible interleaved action)

$=$

$(\alpha.\gamma 1 + \kappa 1.\alpha).S + (.. + \gamma 2.\beta)\ S$

$\mid$

$(\gamma 1.\gamma 2.med + .\kappa 1.med)$

($\kappa 2$ is the interaction)

$=$

$(\alpha.\gamma 1 + .\alpha).S + (.. + \gamma 2.\beta).S$

$\mid$

$(\gamma 1.\gamma 2.med + ..med)$

($\kappa 1$ is the interaction)

$=$

$(\alpha.\gamma1 + ..).S + (.. + \gamma2.\beta).S$

|

$(\gamma1.\gamma2.med + ..med)$

(α is an interleaved action)

(5.34)

From which the partial behaviour of the system S: '$(\beta.\kappa2.\kappa1.\alpha).S$' is derived. With $\kappa2=\kappa1=\kappa$ hidden, this behaviour becomes: '$(\beta.\alpha).S$'. Similar considerations lead to the partial behaviour: $(\alpha.\beta).S$. So that the behaviour of S can be expressed as:

$$S = (\alpha.\beta).S + (\beta.\alpha).S \quad (5.35)$$

which is the same as Equation (5.23).

Again the alternative formulation of the processes of Equation 5.33:

$S1 = (\alpha.\gamma1 + \kappa1.\alpha).S1$;

$S2 = (\beta.\kappa2 + \gamma2.\beta).S2$;

and

$med1 = \gamma1.\gamma2.med1$;

$med2 = \kappa2.\kappa1.med2$ (5.36)

and the parallel composition:

$S1|med1|med2|S2$ (5.37)

leads to the same intended service of S as given by Equation (5.34).

[end of example 5.3]

5.3 Functionality Of Behaviour Expressions

The behaviour of the system as defined by Equation (5.5) or Equation (5.23) provides a certain service. The functionality of Equation (5.5) is a minimum and that of

Equation (5.23) is a maximum (Theorem 4.4.3). That means the maximum functionality f of system behaviour can be expressed as:

functionality(S) = maximum functionality($\alpha+\beta$, $\alpha.\beta + \beta.\alpha$)

with

maximum functionality($\alpha+\beta$, $\alpha.\beta + \beta.\alpha$) =

functionality($\alpha.\beta + \beta.\alpha$) = $f(\alpha,\beta) = \gamma$ (say)

(from Theorem 4.4.2).

From which it can be shown that if a pair of concurrent actions participate in a computation then they are also able communicate. Recalling that the precedence relation is a '<' relationship between actions and the communication relation is a '$\leq$' relationship between actions, the following lemma is proved:

Lemma 5.1

A computation by a pair of concurrent actions induces an equivalence class on E which is equal to the equivalence class of communicating actions induced by the pair in E.

Proof:

Let $\alpha \ \chi \ \beta$ in E and suppose, there is some action γ in E s.t.

" '$\alpha<\beta$' *or* '$\beta<\alpha$'" and $f(\alpha,\beta) = \gamma$ (5.38)

$f(\alpha,\beta) = \gamma$ ==> $\alpha|\beta = \gamma$; From Theorem 4.4.2

==> " α *and* $\beta = \gamma$ "

Action γ can be chosen s.t.

" '$\alpha\leq\gamma\leq\beta$' *or* '$\beta\leq\gamma\leq\alpha$' " (5.39)

Therefore,

" '$\alpha\leq\gamma\leq\beta$' *or* '$\beta\leq\gamma\leq\alpha$'" and $f(\alpha,\beta) = \gamma$ (5.40)

$$\Longrightarrow \alpha\mu\beta \text{ from Definition 3.4}$$

$$\Longrightarrow =\mu = =f \tag{5.41}$$

[end of proof]

Which means communication and computation by a pair of actions in E induce the same equivalence class.

Since S satisfies the Partial Constraints (5.4) and (5.22) the process algebraic operator mapping the "*and*" should result in a behaviour expression which yields a functionality given by "$\alpha.\beta$" *or* "$\beta.\alpha$" *or* "γ". That means that the behaviour of S should satisfy the constraint: " 'α precedes β' *or* 'β precedes α'" *or* 'γ' ". That means functionality of S should be given by:

$$(\alpha+\beta).S \text{ 'operator' } (\alpha.\beta + \beta.\alpha).S = \alpha.\beta + \beta.\alpha + \gamma \tag{5.42}$$

where each term on the RHS of Equation (5.42) computes the functionality of S. The symbol 'operator' represents that process algebraic operator, which, when used for composition of the components on the LHS yields the RHS of Equation (5.42).

Mapping the *'and'* of the Partial Constraints (5.4) and (5.22) by the parallel (||) operator and the logical 'and' of 'α *and* β' by the communication (|) operator respectively of ACP, allows Equation (5.42) to be rewritten as follows:

$$(\alpha+\beta) \parallel (\alpha.\beta + \beta.\alpha) = \alpha.\beta + \beta.\alpha + \gamma$$

or

$$(\alpha+\beta) \parallel (\alpha.\beta + \beta.\alpha) = \alpha.\beta + \beta.\alpha + \alpha|\beta \tag{5.43}$$

where the LHS represents the functionality of S and the RHS the possible behaviours of S which compute that functionality.

The intuitive interpretation of this result is that 'internalising' a computation in which two actions α and β can interleave corresponds to hiding the computation from the environment and replacing it by the 'silent' or unobservable action 'γ'.

Indeed, representing the two processes on the L.H.S. of Equation (5.42) as process graphs and composing them gives the process graph for the R.H.S. as shown in Figure 5.6.

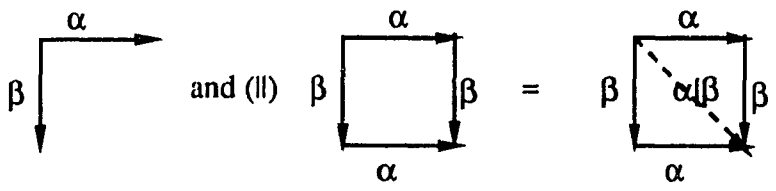


Figure 5.6. Process graphs for parallel composition

Process graphs are directed multigraphs as in [7]. Thus in Figure 5.6 the first process graph representing the process $(\alpha+\beta)$ and the second representing $(\alpha.\beta + \beta.\alpha)$ are parallel composed to give the third process graph on the R.H.S. representing $\alpha.\beta + \beta.\alpha + \alpha|\beta$. The dotted edge representing $\alpha|\beta$ and the edges for $\alpha.\beta$ and $\beta.\alpha$ terminate at the same computational point on the graph. Thus, if $\alpha.\beta + \beta.\alpha$ is hidden from the environment then $\alpha|\beta$ results in the same computation.

The following theorem can now be stated:

Theorem 5.1

Communication and computation equivalence define the same class of actions in a non-deterministic environment.

Proof:

Equation (5.41) of Lemma 5.1 shows that for actions α and β in E:

$\alpha =_f \beta \iff \alpha =_\mu \beta$

[end of proof]

Theorem 5.2

The functionality of the behaviour of a distributed process is given by the composition of the functionalities of distributed and non-distributed parts.

Proof:

The LHS of Equation (5.42) typically, represents the composition of the functionalities of distributed and non-distributed parts of a distributed system S. The functionality of S is given by the functionality of the behaviour 'γ' (or $\alpha|\beta$) or $\alpha.\beta$ or $\beta.\alpha$ on the R.H.S.

This means that whenever the actions α and β participate in a computation in 'parallel' then the functionality of the computation is given by the functionality of the RHS of Equation (5.43). As a consequence the behaviour on the RHS satisfies the constraint " '$\alpha < \beta$' *or* '$\beta > \alpha$' *or* 'α *and* β' ". The same is true for participating actions of communicating processes.

[end of proof]

Corollary 1.

A process algebraic system defining communication in terms of interleaving semantics is not closed under parallel composition (||).

Proof:

In Equation (5.43), $\alpha|\beta \Longrightarrow \alpha =_{\mu} \beta \Longrightarrow \alpha =_{f} \beta \Longrightarrow \alpha \parallel \beta$, from which by Definition 4.4.3:

$$\alpha \parallel \beta = \alpha.\beta + \beta.\alpha + \alpha|\beta$$

$$= \alpha.\beta + \beta.\alpha + \alpha||\beta \qquad (5.44)$$

which shows that the || operator is divergent. And therefore a process algebra with interleaving semantics is not closed under parallel composition.

[end of proof]

Replacing $\alpha|\beta$ by γ in Equation (5.43) gives:

$$\alpha \parallel \beta = \alpha.\beta + \beta.\alpha + \gamma \qquad (5.45)$$

and if δ represents the encapsulation of communication between α and β i.e. $\delta(\alpha \parallel \beta) = \gamma$ then the observable part of Equation (5.45) can be written as:

$$\alpha \parallel \beta = \alpha.\beta + \beta.\alpha \qquad (5.46)$$

where γ is not part of the observable behaviour description.

Also, comparing Equations (5.43) and Equation (5.44) gives:

$$(\alpha+\beta) \parallel (\alpha.\beta + \beta.\alpha) = \alpha \parallel \beta \qquad (5.47)$$

Recall that the observable system behaviour alternately computes the minimum and maximum functionalities. And the composition of the two behaviours in Equation (5.47) satisfies both the non-deterministic and the concurrency constraints, but at different depths of determination. Therefore:

$$(\alpha+\beta) \parallel (\alpha.\beta + \beta.\alpha) = \alpha \parallel \beta$$

$$= (\alpha+\beta) \text{ determination depth one and}$$

$$= (\alpha.\beta + \beta.\alpha) \text{ determination depth two} \qquad (5.48)$$

5.4 Partial Constraints And Design Criteria

The relation between partial constraints and behaviour specifications in a generic process algebra follows. Design criteria are deduced from these definitions.

5.4.1 Constraints And Algebraic Specifications

Recalling that partial constraints are a system of modal logical assertions on the actions of a given communication environment E, the following definitions are stated:

Assumption 5.1 A 'fair' Environment E

A computational environment E is a 'fair' environment iff:

For actions $\alpha,\beta \in E$:

(i) "α precedes β" is true ==> the occurrence of α means that β will eventually occur next (i.e. the occurrence of α means that an action will eventually follow α, and that action is β). The precedence constraint is a *strong fairness constraint.*

(ii) "α *or* β" is true ==> the occurrence of α (or β) means that β (or α) will eventually occur. The "*or*" constraint is a *weak fairness constraint..*

[end of assumption 5.1]

The strong and weak fairness constraints on two actions (α,β) of an environment E can be distinguished by means of a determination of depth two on E. In the case of satisfaction of a strong fairness constraint, a depth two determination *must* always yield the action α followed by the action β. While in the case of satisfaction of a weak fairness constraint, a depth two determination *may* yield the action α followed by the action β or the action β followed by the action α.

This assumption implies that mapping "strong" and "weak" constraints onto the behaviour expressions of a given process algebra lead to the following definition:

Definition 5.1

(i) If B is an action prefix behaviour expression satisfying a "strong" partial constraint, with $B = \alpha.\beta$, and if α is true (occurs) now, then β must eventually be true and if β is true (occurs) now then α is also true. That is the truth of α precedes that of β.

(ii) If B is choice behaviour expression satisfying a "weak" partial constraint, with $B = \alpha \,||\, \beta$, and if α is true (occurs) now, then β will eventually be true, and inversely, if β is true (occurs) now then α will eventually be true.

[end of Definition 5.1]

Definition 5.2

In a communication environment E, the actions in behaviour expressions and partial constraints are self-referential. We let Pc be the smallest set of well-formed partial constraints pc (Definition 4.2) s.t.:

(i) If $pc \in Pc$ then $\neg pc \in Pc$

(ii) If $pc(i) \in Pc \; \forall \; i \in$ some countable set I, then $\cap\, pc(i) \in Pc$

Now, let Π be a set of behaviour specifications in a given behaviour model (process algebra), and for each $\alpha \in E$, let $\Pi \times \Pi \supseteq$ '--α-->' be a relation s.t. for $B \in \Pi$ the

set { B|B --α--> B' and B==>B'} is countable, where the notation B==>B' implies the composition of (one or more) relations such as:

'--α-->' with ' --β-->' with '--γ-->' ...

Pc is interpreted by defining the satisfaction relation |= Π x Pc as follows by induction on the structure of Pc:

(i) The behaviour expression B0 (inaction) |= ' ' (the empty pc)

(ii) B |= pc ==> B satisfies pc (always)

(iii) B |= ¬pc iff not B |= pc

(iv) B |= ∩ pc(i) iff B |= pc(i) ∀ i ε I, the set of integers

(v) If B1 |= pc1 and B2 |= pc2 iff B = B1 'op' B2 |= pc1 'o' pc2

Where the partial constraint operators o are related to the operators: 'op' of the process algebra under consideration as follows:

If 'op' = - non-deterministic or- then, 'op' |= 'or';

If 'op' = - sequentiality-, 'op' |= 'precedence';

If 'op' = - parallel composition-, 'op' |= *'and'*;

If 'op' = - inaction-, 'op' |= ' ';

The operators '∀', '∃' and 'not' are indirectly related to the operators of the process algebra as seen below. Other operators of the process algebra are not considered.

(vi) The modal operators □ and ◊ are defined such that:

(a) B|= pc <==> B |= □pc

(b) If B |= □pc ≡ ∀ B', such that B==>B', then B'|=pc

(c) If B |= ◊pc ≡ ∃ a B1,B2 and pc1,pc2 in E such that:

B = (B1 'op' B2) and (pc1 'o' pc2) ==> pc,

and B1|=pc1 and B2 |=pc2.

The implication '==>' is according to the inference structure of partial constraints and FOPL.

[end of Definition 5.2]

With CCS as the process algebra considered, the following are examples of partial constraints satisfied by CCS expressions:

Example 5.4

(i) $(\alpha+\beta) \models \Box'\alpha$ *or* β'

(ii) $(\alpha+\beta) \models \Diamond'\alpha'$ and $(\alpha+\beta) \models \Diamond'\beta'$

(ii) $(\alpha.\beta) \models \Box'\alpha$ precedes β', then $(\alpha.\beta) \models \Diamond'\beta'$.

Since $(\alpha.\beta) \text{ --}\alpha\text{--> } \beta$ and $\beta \models '\beta'$

(iii) $(\alpha|\beta) \models \Box'\alpha$ *or* β'

(If α,β are not complimentary labels, i.e. they do not communicate)

(iv)$(\alpha|\beta) \models \Diamond'\alpha$ *and* β' is true

(If α,β are complimentary labels)

(v) $\alpha.\beta.\gamma|\beta.\gamma.\alpha \Longrightarrow$ deadlock $\models \Box' ' \Longrightarrow$(assuming full synchronisation)

[end of example 5.4]

5.5 Design And Specification Using Partial Constraints

Recall the first design objective is that of providing a service to users. The service may appear to be distributed or monolithic to users. The users themselves may be distributed (spatially separated) or there may be users at a single location of the distributed system. The former is the case in the lower six layers of the OSI reference model and the latter typifies a user of the OSI application layer. An example of users of the former type are user session layer protocol entities which use a transport service provided by the transport layer. A user transferring a file from a remote file server by using application services such as that of the message handling system X.400, is an example of a user of the latter type.

The second design objective is that of describing the behaviour of the protocol which uses a given underlying service in order to provide the desired service to the users. System design must take into account the joint perspectives of the environment E, which includes users of the types mentioned above, observers and undefinable agents which are the source of non-determinism, but which nevertheless may affect system behaviour, and the system itself. Design activity must result in a behaviour specification which is consistent with this conjoint view.

5.5.1 Design Criteria

5.5.1.1 Partial Constraint Construction

Partial constraints are constructed by simple canonical transformations of given requirement descriptions for each process on a requirement by requirement basis. The partial constraints on the actions of the processes have the atomic forms of relationships between actions defined in chapter 4. viz: the non-deterministic relationship, the concurrent relationship, the communication relationship and the computation relationship. The rules for constructing well-formed partial constraints are also given in Chapter 4. The transformation rules from requirements to partial constraints are assumed to be a user defined simple set of consistent conventions. In the examples that follow, such simple conventions are assumed and easily inferred from the partial constraint constructed.

Given requirements, guidelines to aid in the construction of partial constraints are explained.

In the first step, architectural constraints on the distributed system are identified.

5.5.1.2 Architectural Constraint Construction

It is assumed that all the actions related to the requirements targetted for specification are defined by the set of partial constraints constructed. The guidelines given below aid the architectural design of a distributed system:

A distributed system S and its component processes are identified. The system S consists of a protocol process P distributed into component processes 'PE1' and 'PE2' (say), and a non-distributed process 'med' (say) providing the underlying communication service. The process 'med' may itself consist of decomposed processes 'med1' and 'med2'. The system S operates in a communication environment E in which user processes called 'usr' (say) use the services of the distributed system.

Consistent interactions between processes are obtained with pcs defined on common actions having the same labels.

1. Partial constraints are constructed for the protocol process P, PE1 and PE2 consisting of the constraints pc, pc1 and pc2 respectively, on a constraint by constraint basis, and given by the:

Partial Constraint

pc = 'pc1 *or* pc2' **(5.49)**

where pc1 and pc2 must satisfy the processes PE1 and PE2 and pc satisfies 'P'. The pcs: pc1 and pc2 must be alternately consistent with the user constraints and the underlying service constraints. Note that the 'or' is the non-deterministic *'or'* which may be refined to the communication relation *'and'* in case true concurrency exists in the system.

2. The non distributed underlying service process 'med' interacts with both PE1 and PE2 of P. Therefore, the constraints 'pcm' (say), must be consistent with pc1 and pc2 above. The decomposition of the underlying service into two or more sub-processes may be intended for unidirectional functionalities, serving either of PE1 and PE2 as initiator of data transfer. If the underlying service constraints are labelled 'pcm', 'pcm1' and 'pcm2' respectively, then they are given by the:

Partial Constraint

pcm = 'pcm1 *or* pcm2' **(5.50)**

where 'pcm1' and 'pcm2' may be independant or dependant on each other. Note that the 'or' is the non-deterministic *'or'* which may be refined to the communication relation *'and'* in case 'pcm1' and 'pcm2' are independant of each other.

3. The user process 'usr' constrained by the constraint 'pcu' obtains services from the processes 'PE1' and 'PE2' of 'P'. It may be distributed into components 'pcu1' and 'pcu2', whose constraints are given by:

Partial Constraint

pcu = 'pcu1 *or* pcu2' **(5.51)**

where 'pcu1' and 'pcu2 should be as independant as possible.

Common actions between processes can be assumed to occur at common interaction points or gates defined for each process intended to interact with other processes in the distributed environment of the system. Thus, 'PE1' and 'PE2' interact with 'user1', 'user2', and with 'med1' and 'med2'. Common interaction points between 'PE1', 'PE2 and 'user1, user2 respectively are defined, and similarly for the other processes.

Additional interaction points can be optionally defined between the usr and the protocol, and the protocol and the medium as a design choice. The additional interaction points serve to differentiate classes of interactions. For example, each process PE1 and PE2 may be defined to have two interaction points in common with the medium to distinguish between the class of 'send' actions and 'receive' actions. An example of such a design choice is given for the protocol design and specification of Section 5.6.

5.5.1.3 Behaviour Specifications From Partial Constraints

Behaviour specifications in the form of guarded recursion equations are constructed to satisfy the partial constraints so obtained, using Definition 5.4.1. The behaviour expressions are recursive because the associated partial constraint must always be satisfied. They are guarded in order to obtain expressions which do not diverge, according to the Recursive Definition Principle (RDP) [7]. All the behaviour specifications are partial, in that the complete set of requirements is not specified.

Assuming that the process algebra CCS is used for behaviour specification, the following formulae are used to construct the recursive processes:

1. The protocol processes P, PE1,PE2 are guarded recursive processes satisfying the partial constraints pc, pc1 and pc2 respectively given by the:

Process Formula

$$P = PE1 + PE2$$

or

$$P = PE1 \parallel PE2 \qquad (5.52)$$

2. The processes 'med', 'med1' and 'med2' of the underlying service are guarded recursion equations satisfying the partial constraints, 'pcm','pcm1' and 'pcm2' respectively. They are given by the:

Process Formula

med = med1 + med2

or

med = med1 || med2 **(5.53)**

3. The processes 'usr', 'user1' and 'user2' are guarded recursive equations satisfying the partial constraints: 'pcu', 'pcu1' and 'pcu2' respectively. They are given by the:

Process Formula

usr = user1 + user2

or

usr = user1 || user2 **(5.54)**

Assuming all the processes of E have been specified with respect to the requirements, the following generic formulae are used to validate and verify the specifications:

Process Formula

S = usr || P || med **(5.55)**

or

S = usr || PE1 || PE2 || med **(5.56)**

Where full synchronisation is assumed for all the actions participating in the parallel (||) composition expression of the Formula (5.55). The Formula (5.55) or a variant of the type given in the Formula (5.56), should result in deadlock-free traces. The variant (5.56) provides deadlock free traces in systems with true concurrency or parallelism. Validation and verification of specifications obtained from the above formulae are discussed in Chapter 6.

The above criteria imply the following general principles of design:

1. A system S is decomposed into pairs of distributed processes. The behaviour of each entity in a pair of processes is independently described.

2. Distribution of the system implicitly assumes a non-distributed component (underlying service). The behaviour of the distributed processes and the underlying service must be consistently specified according to the criteria above. The Formulae (5.54) and (5.55) must yield consistent traces without deadlock.

3. Functionality of the system is obtained from computations derived from the Formula (5.55).

4. Partial constraints on system behaviour are constructed in terms of actions (interactions). System behaviour alternates between the behaviour of distributed components and non-distributed components of the system. i.e between the external (observable by the environment) and the internal (not observable by the environment) behaviour of the system.

5. System behaviour satisfies both external and internal constraints.

6. Internal behaviour specification must be consistent with external behaviour specification and vice versa when evaluated against the constraints satisfied by the specification.

7. The externally observable behaviour of the system, viz: that of the distributed system components with user processes in the environment, is in a communication relation. In other words the process 'P' and 'usr' are parallel composed to compute communication behaviour.

8. Similarly, internal behaviour of the system, viz: that between 'P' and 'med' is also in a communication relation.

9. System behaviour can be expressed as a closed system of equations, representing external and internal constraints. External Constraints are those satisfied by the process 'P' and the process 'usr'. Internal constraints are those satisfied by the process 'P' and the process 'med'. Actions between 'P' and 'usr' and between 'P' and 'med' pair-wise, are in a communication relationship.

10. All specifications obtained are partial, but can be extended on a requirement by requirement basis. Each requirement, however is completely defined. Mutual consistency of requirements is ensured by the well-formed of the partial constraints associated with the requirements.

Note:The class of specifications generated by partial constraints consists of partial specification such that every specification has a finite (or countable) number of non-deterministic choice expressions. Each component expression however can have infinite

recursion. Therefore, it is decidable whether any given process algebraic specification belongs to this class or not.

5.6 An Example - The Alternating Bit Protocol

To illustrate, A version of the Alternating Bit protocol [6] and the service it provides to users, is specified.

The service provided by this protocol consists of transmission of messages from one user to another remote user. The protocol encodes messages with a binary sequence number, a 0 or a 1. Messages sent are acknowledged by the remote protocol entity. The sender entity alternates between message sequence number values 0 and 1. The receiver entity acknowledges each message with an acknowledgement of the same number. In the version of the protocol specified here, user A (say) sends messages, while user B receives the messages sent by A. The protocol returns an acknowledgement with the same sequence number as the message received.

Note that another version of the protocol can also be specified wherein, the protocol returns an acknowledgement with the sequence number of the next expected message. This differs slightly from the original version of the protocol, in that the acknowledgement of a message with sequence number 0 has the number 1 (instead of 0) and vice versa. This change does not affect the reliability of the protocol, rather it makes it easily extendible to the sliding window protocol. In the sliding window protocol, the acknowledgement always bears the sequence number of the next expected message by the receiver.

The medium is unreliable and can lose messages. Retransmission of lost messages after a timeout period must be handled by the protocol.

The architecture of the AB protocol - medium system is shown in Figure 5.7:

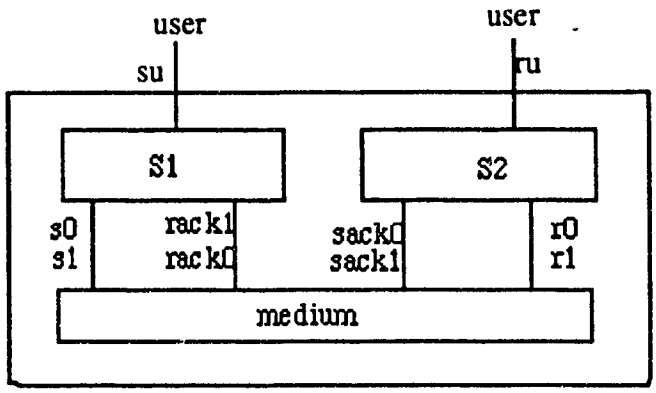


Figure 5.7. The Alternating Bit protocol

For user-protocol interactions:

Let su,ru denote the 'Send' action and 'Receive' action by users.

For protocol medium interactions:

Let s0 and s1 denote the sending of messages with sequence number 0 and 1 respectively and let r0 and r1 denote corresponding receive actions.

For acknowledgements, let sack0 and sack1 represent send actions and rack0 and rack1 corresponding receives.

The correct functioning of the protocol requires that:

1. The protocol send a message with a given sequence number only if it has received the acknowledgement for the previous message sent with the alternate sequence number. For example, rack0 must precede s1.

2. In order that the sender always alternates between the sequence numbers 0 and 1, the partial constraint constraining the sending of s1 (say) must precede that of sending s0.

Therefore, the constraints for the distributed processes are stated as follows:

1. " 'r0 precedes su precedes s0 precedes pcs1'

or

'rack0 precedes su precedes s1 precedes pcs0' " (5.57)

where

pcs0='rack1 precedes su precedes s0 precedes pcs1'

pcs1= 'rack0 precedes su precedes s1 precedes pcs0'

2. " 'r0 precedes ru precedes sack0 precedes pcr1'

or

'r1 precedes ru precedes sack1 precedes pcr0' " (5.58)

where

pcr0='r0 precedes ru precedes sack0 precedes pcr1'

pcr1= 'r1 precedes ru precedes sack1 precedes pcr0'

The non-distributed constraints are stated as follows:

1. " 's0 precedes r0' *or* 's1 precedes r1' " (5.59)

2. " 'sack0 precedes rack0' *or* 'sack1 precedes rack1' " (5.60)

3. " 'rack0' *or* 'rack1' " (5.61)

Note that constraint (5.61) has been added to satisfy correctness concerns for the initial conditions of the protocol behaviour, where initially rack0 or rack1 is true.

The user constraints are stated as follows:

1. 'su *or* ru' (5.62)

The distributed processes satisfying these constraints are given by:

where

S1 = S10 + S11 (5.63)

and

S2 = S20 + S21 (5.64)

where

S10 = rack1.su.s0.S11

S11 = rack0.su.s1.S10

S20 = r0.ru.sack0.S21

S21 = r1.ru.sack1.S20

The process S10 recurses to S11 and vice-versa. Similarly, S20 to S21 and vice versa.

The non-distributed processes satisfying the non-distributed constraints are given by:

med = med1 + med2 (5.65)

where

med1=s0.r0.med + s1.r1.med

med2 = sack0.rack0.med + sack1.rack1.med + rack0.med + rack1.med

and the user processes are given by:

usr = user1 + user2 (5.66)

where

user1 = su.usr

user2 = ru.usr

The composition of all the processes using the Formula (5.55) gives:

S1|S2 | (med1 + med2) | usr

=

rack0.su.s1.S10 + rack1.su.s0.S11

|

r0.ru.sack0.S21 + r1.ru.sack1.S20

|

s0.r0.med + s1.r1.med

+

sack0.rack0.med + sack1.rack1.med + rack0.med + rack1.med

|

su.med + ru.med

=

.su.s1.S10 + rack1.su.s0.S11

|

r0.ru.sack0.S21 + r1.ru.sack1.S20

|

s0.r0.med + s1.r1.med

+

sack0.rack0.med + sack1.rack1.med + .med + rack1.med

|

su.usr + ru.usr

(rack0 is an interaction)

=

..s1.S10

|

r0.ru.sack0.S21 + r1.ru.sack1.S20

|

s0.r0.med + s1.r1.med

+

sack0.rack0.med + sack1.rack1.med + rack0.med + rack1.med

|

.usr + ru.usr

(su is an interaction)

=

...S10

|

r0.ru.sack0.S21 + r1.ru.sack1.S20

|

s0.r0.med + .r1.med

+

sack0.rack0.med + sack1.rack1.med + rack0.med + rack1.med

|

su.usr + ru.usr

(s1 is an interaction)

=

rack1.su.s0.S11

|

.ru.sack1.S20

|

..med

|

su.usr + ru.usr

(r1 is an interaction)

=

rack1.su.s0.S11

|

..sack1.S20

|

s0.r0.med + s1.r1.med

+

sack0.rack0.med + sack1.rack1.med + rack0.med + rack1.med

|

su.usr + .usr

(ru is an interaction)

=

rack1.su.s0.S11

|

...S20

|

s0.r0.med + s1.r1.med

+

sack0.rack0.med + .rack1.med + rack1.med + rack1.med

|

su.usr + ru.usr

 (sack1 is an interaction)

=

.su.s0.S11

|

r0.ru.sack0.S21

|

..med

|

su.usr + ru.usr **(rack1 is an interaction)**

=

..s0.S11

|

r0.ru.sack0.S21

|

s0.r0.med + s1.r1.med

+

sack0.rack0.med + sack1.rack1.med + rack0.med + rack1.med

|

.usr + ru.usr **(su is an interaction)**

=

...S11

|

r0.ru.sack0.S21

|

.r0.med + s1.r1.med

+

sack0.rack0.med + sack1.rack1.med + rack0.med + rack1.med

|

su.usr + ru.usr **(s0 is an interaction)** (5.67)

Note that the interaction in boldface is a possible interaction, indicated

in the equation by replacing the corresponding action of the interacting processes by a dot. However, once the first action of a recursive term participates in an interaction, then the next possible interaction must be the one following that action.

So that a possible behaviour trace derived from (5.67) is the:

Trace

S |= <rack0,su,s1,r1,ru,sack1,rack1,su,s0,S10> (5.68)

Similarly, we can obtain the:

Trace

S |= <rack1,su,s0,r0,ru,sack0,rack0,su,s1,S11> (5.69)

where S10 and S10 give traces appropriate to their definitions.

No other deadlock free traces can be obtained from the composition.

The traces permit the construction of corresponding partial constraints which are satisfied by system behaviour expressed as:

S = rack1.su.s0.r0.ru.sack0.rack0.su.s1.S11

+ rack0.su.s1.r1.su.sack1.rack1.su.s0.S10 (5.70)

Note that the actions rack1 and rack0 initially are 'dummy' actions to generate the correct recursive behaviour of the protocol.

Alternatively, the behaviour definitions:

S1 = rack0.su.s1.S10 + rack1.su.s0.S11

S2 = r0.ru.sack0.S21 + r1.ru.sack1.S20 (5.71)

where S21 = r1.ru.sack1.S20 and S20 = r0.ru.sack0.S21

The non-distributed processes satisfying the non-distributed constraints are given by:

med1 = s0.r0.med1 + s1.r1.med1

med2 = sack0.rack0.med2 + sack1.rack1.med2

+ rack0.med2+rack1.med2 (5.72)

and the user processes defined as before with the parallel composition:

S = S1|med1|med2|S2|user1|user2 (5.73)

will derive the same service for the system S.

Retransmission On Timeout

To specify behaviour which includes retransmission on timeout due to lost messages the following constraints must be satisfied:

The constraints for the distributed processes are stated as follows:

Pc12:

1. " 'rack0 precedes su precedes s1 · ·ecedes tstart1 precedes

('tout1 precedes s1 precedes tstart' *or* pc11)'

or

'rack1 precedes su precedes s0 precedes tstart0 precedes

('tout0 precedes s0 precedes tstart0' *or* pc10)' " (5.74)

where

pc10 = 'rack1 precedes su precedes s0 precedes tstart0 precedes

('tout0 precedes s0 precedes tstart0' *or* pc10)' "

pc11 = 'rack0 precedes su precedes s1 precedes tstart1 precedes

('tout1 precedes s1 precedes tstart' *or* pc11)'

Note that the constraint within the quotes in the parenthesis is a nested constraint. The constraint is defined upto depth 7. No further weakening of constraints (from 'precedes' to *'or'* is possible).

2. " 'r0 precedes ru precedes sack0 precedes pcr1'

or

'r1 precedes ru precedes sack1 precedes pcr0' " (5.75)

where

pcr0='r0 precedes ru precedes sack0 precedes pcr1'

pcr1= 'r1 precedes ru precedes sack1 precedes pcr0'

The constraints (5.74) and (5.75) show that the sending of messages with sequence numbers 1 and 0, as well as their reception are mutually constrained due to alternation in the two protocol procedures. Therefore their behaviour is defined with mutual recursion on processes S10,S11 as before:

S1 = S10 + S11 (5.76)

where

S10 = rack1.su.s0.tstart0.(tout0.s0.tstart1.L0 + S11)

S11 = rack0.su.s1.tstart1.(tout1.s1.tstart1.L1 + S10)

with L0 = tout0.s0.tstart1.L0 and L1 = tout1.s1.tstart1.L1

S2 = S20 + S21 (5.77)

where

S20 = r0.ru.sack0.S21

S21 = r1.ru.sack1.S20

The constraints related to the timer are:

2. " 'tout0' *or* 'tstart0' "

or

" 'tout1' *or* 'tstart1' " (5.78)

where, for the timer: the actions tout0, tout1, tstart0 and tstart1 represent timeout and timestart for messages with sequence number 0 and 1 respectively.

The timer process is specified as follows:

T = tstart0.T + tout0.T + tstart1.T + tout1.T (5.79)

The non-distributed constraints are:

1. " 's0 precedes ('r0 *or* s0')' *or* 's1 precedes ('r1 *or* s1')' " (5.80)

2. " 'sack0 precedes (rack0 *or* sack0)'

or 'sack1 precedes (rack1 *or* sack1)' " (5.81)

3. " 'rack0' *or* 'rack1' " ; (initial conditions) (5.82)

with the behaviour of the medium:

med = med1 + med2

med1 = s0.(r0 + s0).med + s1.(r1 + s1).med

med2 = sack0.(rack0 + sack0).med + sack1.(rack1 + sack1).med

$$+ \text{rack0.med} + \text{rack1.med} \tag{5.83}$$

so that the composition of the behaviours:

$$S1|S2|(\text{med1} + \text{med2})\,|\,T\,|\,\text{usr}$$

$$\begin{aligned}
&|= \text{rack0.su.s1.tstart1.(tout1.s1.tstart1.L1} + \text{S10)} \\
&\quad + \\
&\quad \text{rack1.su.s0.tstart0.(tout0.s0.tstart0.L0} + \text{S11)} \\
&\quad | \\
&\quad \text{r0.ru.sack0.S21} + \text{r1.ru.sack1.S20} \\
&\quad | \\
&\quad \text{s0.(r0} + \text{s0).med} + \text{s1.(r1} + \text{s1).med} \\
&\quad + \\
&\quad \text{sack0.(rack0} + \text{sack0).med} + \text{sack1.(rack1} + \text{sack1).med} \\
&\qquad + \text{rack0.med} + \text{rack1.med} \\
&\quad | \\
&\quad \text{tstart0.T} + \text{tout0.T} + \text{tstart1.T} + \text{tout1.T} \\
&\quad | \\
&\quad \text{su.usr} + \text{ru.usr}
\end{aligned}$$

$$\begin{aligned}
&= \text{.su.s1.tstart1.(tout1.s1.tstart1.L1} + \text{S10)} \\
&\quad + \\
&\quad \text{rack1.su.s0.tstart0.(tout0.s0.tstart0.L0} + \text{S11)} \\
&\quad | \\
&\quad \text{r0.ru.sack0.S21} + \text{r1.ru.sack1.S20} \\
&\quad | \\
&\quad \text{s0.(r0} + \text{s0).med} + \text{s1.(r1} + \text{s1).med} \\
&\quad +
\end{aligned}$$

sack0.(rack0 + sack0).med + sack1.(rack1 + sack1).med

+ .med + rack1.med

|

tstart0.T + tout0.T + tstart1.T + tout1.T

|

su.usr + ru.usr

(rack0 is the interaction)

= ..s1.tstart1.(tout1.s1.tstart1.L1 + S10)

|

r0.ru.sack0.S21 + r1.ru.sack1.S20

|

s0.(r0 + s0).med + s1.(r1 + s1).med

+

sack0.(rack0 + sack0).med + sack1.(rack1 + sack1).med

+ .med + rack1.med

|

tstart0.T + tout0.T + tstart1.T + tout1.T

|

.usr + ru.usr

(su is the interaction)

= ...tstart1.(tout1.s1.tstart1.L1 + S10)

|

r0.ru.sack0.S21 + r1.ru.sack1.S20

|

s0.(r0 + s0).med + .(r1 + s1).med

+

sack0.(rack0 + sack0).med + sack1.(rack1 + sack1).med

+ rack0.med + rack1.med

|

tstart0.T + tout0.T + tstart1.T + tout1.T

|

su.usr + ru.usr

(s1 is an interaction)

= ....(tout1.s1.tstart1.L1 + S10)

|

r0.ru.sack0.S21 + r1.ru.sack1.S20

|

s0.(r0 + s0).med + .(r1 + s1).med

+

sack0.(rack0 + sack0).med + sack1.(rack1 + sack1).med

+ rack0.med + rack1.med

|

tstart0.T + tout0.T + .T + tout1.T

|

su.usr + ru.usr

(tstart1 is an interaction)

= ....(tout1.s1.tstart1.L1 + S10)

|

r0.ru.sack0.S21 + .ru.sack1.S20

|

s0.(r0 + s0).med + .(+ s1).med

+

sack0.(rack0 + sack0).med + sack1.(rack1 + sack1).med

+ rack0.med + rack1.med

|

tstart0.T + tout0.T + tstart1.T + tout1.T

|

su.usr + ru.usr

(r1 is an interaction)

= ....(tout1.s1.tstart1.L1 + S10)

|

..sack1.S20

|

s0.(r0 + s0).med + .(+ s1).med

+

sack0.(rack0 + sack0).med + sack1.(rack1 + sack1).med

+ rack0.med + rack1.med

|

tstart0.T + tout0.T + tstart1.T + tout1.T

|

su.usr + .usr

(ru is an interaction)

= ....(tout1.s1.tstart1.L1 + S10)

|

...S20

|

s0.(r0 + s0).med + s1.(r1 + s1).med

+

sack0.(rack0 + sack0).med + .(rack1 + sack1).med

+ rack0.med + rack1.med

|

tstart0.T + tout0.T + tstart1.T + tout1.T

|

su.usr + ru.usr

(sack1 is an interaction)

= .su.s0.tstart0.(tout0.s0.tstart0.L0 + S11)

|

...S20

|

s0.(r0 + s0).med + s1.(r1 + s1).med

+

sack0.(rack0 + sack0).med + .(+ sack1).med

+ rack0.med + rack1.med

|

tstart0.T + tout0.T + tstart1.T + tout1.T

|

su.usr + ru.usr

(rack1 is an interaction)

= .su.s0.tstart0.(tout0.s0.tstart0.L0 + S11)

|

...S20

|

s0.(r0 + s0).med + s1.(r1 + s1).med

+

sack0.(rack0 + sack0).med + .(+ sack1).med

+ rack0.med + rack1.med

|

tstart0.T + tout0.T + tstart1.T + tout1.T

|

su.usr + ru.usr

(su is an interaction)

(5.84)

and the next interaction is s0. So that it is easy to see that the following traces can be obtained:

<rack0 , su , s1 , tstart1 , r1 , ru , sack1 , rack1 , su , s0...

or

rack0 , su , s1 , tstart1 , tout1 , s1 , tstart1 , tout1 , s1 , tstart1...>

(5.85)

and a similar set of traces starting with rack1.

Obtaining the associated constraints from (5.85), the following behaviour expression for the system can be deduced:

S = rack0.su.s1.tstart1.(tout1.s1.tstart1.L1 + r1.ru.sack1.S10')

+

rack1.su.s0.tstart0.(tout0.s0.tstart0.L0 + r0.ru.sack0.S11')

(5.86)

where

L0 = tout0.s0.tstart0

L1 = tout1.s1.tstart1

S10' = rack1.su.s0.tstart0.(tout0.s0.tstart0.L0 + r0.ru.sack0.S11')

S11' = rack0.su.s1.tstart1.(tout1.s1.tstart1.L1 + r1.ru.sack1.S10')

Note that there are no other traces possible.

The advantages of this method of design are that this specification is tractable and lends itself to validation of the service and verification of protocol behaviour properties, as we shall see in the next chapter.

Be sure of it, give me the ocular proof.

...

Make me to see't, or at the least so prove it
That the probation(proof) bear no hinge nor loop
To hang a doubt on,...

Othello, Act II, Scene iii, (lines 360-366)

Chapter 6

DISTRIBUTED SYSTEM SPECIFICATION, VALIDATION AND VERIFICATION

The objective of validation and verification activity in the design-to- specification trajectory is to gain a measure of confidence in the formal specification of a distributed system. The partial constraint method of design and specification developed in the previous chapters is used validation and verification of distributed systems. A method for validation and a method of verification is developed in this chapter. Both methods can be used in the design-to-specification phase of system development. In this chapter, a simplified OSI Transport protocol class(0) is validated and the Alternating Bit protocol is verified to illustrate the methods. The Transport protocol is validated against required service. Some safety and liveness properties of the Alternating Bit protocol specified in Chapter 5 are verified. The specifications used for validation and verification are partial specifications. Specifications are considered to be partial if only a subset of the entire set of requirements are used for specification purposes. That means that the specifications do not necessarily define the complete set of requirements. Therefore, the validation and verification method presented in this thesis demonstrates correctness of partial specifications only. This implies that a specification is correct with respect to a relevant subset of the requirements. The proof technique uses the modal operators defined in Definition 5.4.1.

The OSI transport service obtained from the composition of the protocol entities and the underlying service is specified as an example of a real distributed system, and then validated against correct intentions. Verification is illustrated by means of the classic example of a 'real' protocol - the Alternating Bit protocol.

6.1 Validation And Verification

Software validation and verification techniques are based on Floyd's notion of program correctness [27]. Some well known methods include the method of inductive assertions [27], the method of invariants [36], proof of equivalence of assertions on hierarchical programs [68], subgoal induction [58] and predicate transformations [83].

Nearly every method involves the formal definition of assertions either as mathematical predicates or assertions in a formal language created for this purpose. Proof of consistency between program behaviour with respect to a given logical property, stated as an assertion is considered to be verification of the program with respect to that logical property.

Verification methods analogous to program verification methods have also been applied to distributed systems. Verification of distributed systems are based on proving assertions defined on system states [85], in a programming language [76] or temporal logic [31]. Verification by transformation of formal system specifications based on notions of behavioural equivalence has been discussed in [72]. Verification during the design and specification phase has been demonstrated in [21].

The underlying motivation for validation and verification of system behaviour is correctness concerns. The issue of correctness of distributed system behaviour is analogous to the issue of correctness of program behaviour.

6.1.1 Program Correctness

"A program is considered to be correct if program output satisfies output requirements for every input specified by the input requirements" - [67] . For inputs consistent with requirements, correct program behaviour should result in output satisfying output requirements.

The concept of program correctness was first defined in [27]. A program P is correct with respect to an input assertion ϕ and an output assertion ψ if it can be proved that for ϕ true at the start of a program, ψ is true at its termination. This view encapsulates the most important notions of verification. Given ϕ true at the start of the program Floyd used a method called the method of inductive assertions to prove the truth of the assertions ψ.

6.1.2 Validation Of Partial Specifications

Recall from Chapter 5, that the class of partial specifications generated by the partial constraint method of specifications generates specifications which have only a finite (or countable) number of non-deterministic choices. This means although individual traces generated by a given partial specification may correspond to infinite behaviour, each partial specification can generate only a finite (or countable) set of such traces.

For distributed system behaviour, validation implies a demonstration that the system behaviour as specified is of correct intention. That means that system behaviour has been consistently specified - syntactically with respect to the syntax of the specification language used, and semantically with respect to intentions. Syntactic consistency can be validated by static syntax checking methods. Syntactic consistency will not be considered here. Semantic consistency of the system can be demonstrated by a consistent composition of the system's component descriptions. For example, consistency of a protocol's behaviour can be shown by composition of protocol components with the underlying service. For distributed systems, semantic consistency can be demonstrated either, by composition of the system with the users (user processes) in the environment, (as we shall see) or, by comparison of system service with intended service. Simulation of the behaviour of the composed system is often used to validate a system.

Validation Using Partial Constraints

Behaviour simulation of composed processes, is based on the inference rules for composition given by the semantics of the process algebra used. If composition of system components yields deadlock-free observable traces, the specification can be interpreted to be semantically consistent. A deadlock is detected during trace recording if at any point in the computation, the next action (which is not a successful termination action) cannot occur and the remainder of the system behaviour satisfies the empty partial constraint. Using the observed traces as the basis of determinations (see Chapter 4, Assumption 4.1) made on the actions of the communication environment, associated partial constraints can be constructed. And if system services as interpreted from the partial constraints are consistent with intended services, then the specification is semantically validated. Or, if composition of system components with user processes, for which the service is intended

yields deadlock free traces, then the service is consistent with intentions and the specification is semantically validated.

System service requirements are semantically analogous to the assertions ψ, above. If system behaviour as specified, can be demonstrated to be consistent with service requirements, then the specification is validated to be of correct intent. The specifications of system components from which valid behaviour can be inferred, are thereby also validated.

In the partial constraint method of validation explained below, simulation is the basis of partial constraint determinations (Assumption 4.1). The depths of determinations are assumed to be finite. Partial constraints are used to define a partial specification of the target system and user processes. Composition of all the processes should result in observable deadlock-free traces of behaviour. The traces so obtained form the basis of a systematic mode of determining partial constraints on behaviours in E at a predefined depths of determinations (see Definition 4.7). Behaviour expressions satisfying partial constraints (Definition 5.2) are constructed. Consistency of the behaviour expressions so obtained with service requirements of the system validates the system. Note that the validation results are true only for finite depths of determinations.

6.1.3 Verification Of Partial Specifications

Verification of distributed system behaviour is analogous to proving logical properties of program behaviour. A verification method analogous to the Floyd-Hoare method of proving program correctness is developed below. Partial constraints are used as assertions in the verification process described below. In the partial constraint method of verification, the assertion ϕ is replaced by the partial constraint pc_i (corresponding to input

assertions on input requirements), and ψ is replaced by the partial constraint PcO (corresponding to output assertions on output requirements). The logical property to be verified is represented in terms of the partial constraints pcI and pcO. So that, if pcI is true at the start (instantiation) of system behaviour then pcO must be true at its termination, if the behaviour terminates.

Verification Method

The verification method proceeds as follows:

1. Partial constraints for a required logical property to be verified are constructed from requirement statements of that property. This corresponds to determinations (Assumption 4.4.1) made on the environment E of the system. The sets of partial constraints so obtained are assumed to correspond to input assertions pcI (ϕ) and output assertions pcO (ψ) for the processes whose behaviour is to be verified.

2. For a given process, if the associated partial constraint pcI is true at process instantiation (the start of system behaviour), and pcO is true at its termination (if it terminates), then the logical property is interpreted as being verified.

3. To verify a given component process of the distributed system, it is composed in parallel with all other component processes and user processes in the environment. The environment is assumed to be 'fair' (by Assumption 5.1 in Chapter 5).

4. The set of all possible traces of behaviour are derived from the composition. This is possible because the specifications are partial, yielding a finite (or countable) trace set. Only finite prefixes of each infinite trace are considered.

5. Deadlock-free traces yield partial constraints which hold true for system behaviour at finite depths of determinations. The set of partial constraints so obtained when composed together yield pcO.

6. PcO true implies that the property represented by (pcI,pcO) obtained from (ϕ,ψ) is verified.

The two types of logical properties to be verified are Safety and Liveness. Safety properties properties generally specify that: "nothing bad will happen". That means if some behaviour occurs it will be safe behaviour (corresponding to required behaviour). Liveness properties specify that: "Only good things will happen". That means useful behaviour (corresponding to desired behaviour) will eventually happen.

System behaviour is considered to be safe, with respect to ϕ and ψ, if and when, the behaviour terminates. Proof that indeed, the behaviour will eventually terminate makes the safety property a liveness property.

6.1.4 Safety And Liveness

Proving safety and liveness properties amounts to proving the following:

Safety

Given a functional statement of a safety property, proving safety implies that system behaviour if it terminates, terminates successfully. Successful termination means that on termination the partial constraint pcO (or the assertion ψ) is true.

Safety Properties Of Finite State Transition Descriptions

For state transition descriptions, if there exist transition paths, with no intervening deadlock states, in which ϕ is true at the initial state and ψ is true at the final state, then the system is considered to be able to operate safely with respect to the given property.

Deadlock states are non-final states which prevent control from reaching the final state along a transition path. States (or set of states) from which there are no transitions leading out are deadlock states. That is, in traversing deadlock states control either stops abnormally or loops indefinitely. System safety with respect to the property under consideration implies that, there exist safe transition paths through which the system can traverse. However, there is no guarantee that there may be other paths which have deadlock states or infinite loops when they are expected to be terminating paths. Note that termination is not proved for safety properties, rather the truth of assertions in case of termination, is proved.

For non-terminating systems, if ϕ and ψ are replaced by an invariant I (say), and I is true each time control reaches the beginning of a looping transition path, (without intervening deadlock states) then safety is verified.

Safety Properties Of Algebraic System Descriptions

For algebraically specified systems, if the partial constraint pcI (ϕ) is true at a given behaviour instantiation, and if pcO (ψ) is true at behaviour termination (i.e.there exists an observable trace - with no intervening deadlocks), the system is interpreted to be safe with respect to the logical property represented by pcI and pcO.

Conditions For Deadlock

Deadlock occurs if any action of a component process is unable to synchronise (or communicate) with an action in the environment. Or, if all the processes under composition wait forever for synchronisation. Deadlock causes the behaviour to terminate abnormally preventing control from reaching action(s) after which the behaviour is considered to terminate successfully.

For non-terminating processes, safe operation must be proved on the basis of every finite prefix of an infinitely recurring trace expression. Partial constraints are constructed on the basis of determinations of finite depths made on sets of the potentially infinite traces. Deadlock causes the behaviour to terminate abnormally, preventing control from reaching action(s) which guard re-instantiation of recursive processes. If ϕ and ψ (or I) is true before the initial action on behaviour instantiation, and is also true for the recursive instantiation of that behaviour (without intervening deadlocks), then safety is verified. Note that safe operation of recurrent behaviour implies that every finite prefix of an infinite trace is safe.

Liveness

Proving liveness means proving that system behaviour progresses. That means the system performs useful work at every step in its behaviour. That amounts to proving eventual successful termination, or that eventually the system executes useful behaviour.

The Liveness Property Of State Transition Descriptions

For state transition systems, if it can be shown that ϕ is true at the initial state, and that *every* transition path leads to the final state, without intervening deadlocks, i.e. the behaviour eventually terminates with ψ true, then the system is livelock free.

For non-terminating systems, if it can be shown that ϕ and ψ (or I) is true at the initial state, and that *every* transition path loops back to the initial state with I holding true, without intervening deadlocks, i.e. the behaviour eventually recurs with I true, then the system is livelock free.

The Liveness Property Of Algebraic System Descriptions

For algebraically specified terminating behaviour, if ϕ is true at behaviour instantiation, and the behaviour eventually terminates, without intervening deadlocks, with ψ true, then the system is live with respect to ϕ and ψ.

For non-terminating algebraically specified systems, proving that the system will eventually recurse with ϕ and ψ (or I) true, verifies system liveness. However, complete algebraic specifications of protocols and distributed systems usually exhibit non-terminating behaviour with potentially infinite number of infinite traces. This makes verification difficult. Complete protocol and distributed system specifications based on the algebraic model generate infinite transitions systems. Such systems are usually difficult to verify.

Partially Specified Algebraic Systems

In order to make algebraic systems tractable to verification analysis, it becomes necessary to restrict the form and structure of algebraically specified systems. The partial constraint methodology restricts algebraic specifications to a subset of the complete requirement constraints on the behaviour of the system. This restriction is further emphasized with a specification structure limited to non-deterministic choices of guarded recursive equations (see Chapter 4 and 5). Partial constraints lead to partial specifications which are tractable to verification analysis.

For non-terminating partially specified algebraic systems, proving that the system will recurse, with ϕ and ψ (or I) eventually true for all sequences of actions guarding the recursion of a process, verifies system liveness.

In terms of system functionalities, proving safety implies partial correctness and liveness implies total correctness.

Verification of system behaviour implies 'actual' behaviour. In the following, observable traces derived from the composition of behaviour expressions are associated with the 'occurrence' of actions (or behaviour transitions). They are assumed to represent actual behaviour. Transitions or actions occur when common actions of parallel composed processes synchronise.

6.2 Validation Of The Transport protocol

Example 6.1

Consider the Connection establishment phase of the OSI transport layer protocol (class 0). (This example is an extension of the example given in [1],[3]). To keep the presentation of ideas simple and comprehensive, a subset of all the required behaviours is considered. Initially, Disconnect requests or indications are not considered.

Figure 6.1 shows the architecture of the system. The distributed system environment consists of user processes, a protocol and an underlying service. The logical structure of the protocol consists of two protocol entities PE1 and PE2 respectively. The combination of PE1 and PE2 is the protocol process, with two pairs of interaction points t1 and t2, and p1 and p2 respectively. The interaction points t1,t2 are shared with user processes of the environment - usr1 and usr2 respectively. While p1 and p2 are shared with the underlying service.

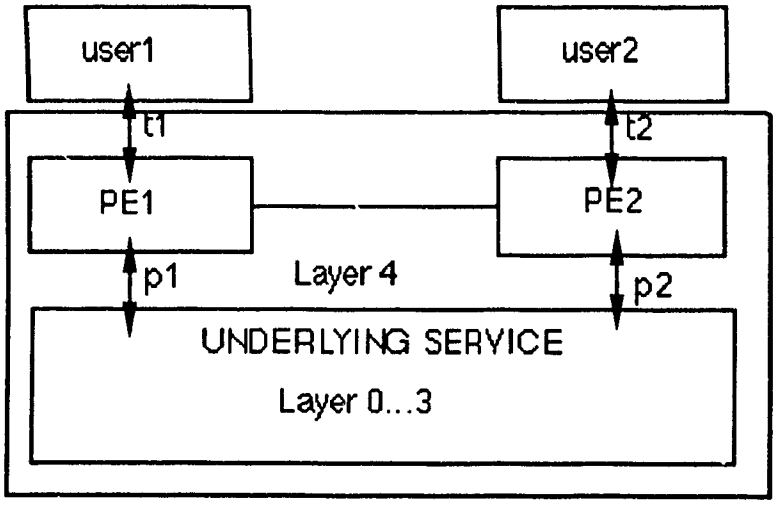


Figure 6.1 Logical structure of the OSI Transport protocol.

It is assumed that the actions of an environment E can be indexed by values of predefined data type sorts, and that the set of values of all data types are countable. For example:

The partial constraint

"p1(send/output)CR TPDU precedes p2(receive/input) CR TPDU"

(6.1)

constraints a generic action p1 (the action occurs at the interaction point p1 of the process) indexed by a value: '(send/output)CR TPDU' to precede a generic action p2 (the action occurs at the interaction point p2 of the process) indexed by any value of the CR TPDU sort. The index values come from countable sorts (for example, the CR TPDU sort), making the actions unique. An index value represents a typical, well defined value of the associated predefined sort. Note that the indexes are not parameters as such, but interpreting them as parameters makes the specification readable. In principle system

behaviour can be specified by a unique labelling of all the actions. But this would make the specification unreadable.

By convention we assign the symbol ! to the '(send/output)' component of the index and ? to the '(receive/input)' component of the index.

A second convention is the same as the matching convention followed in LOTOS for synchronisation of actions. In LOTOS, actions with common generic labels can synchronise if action parameters have the form:

1. ?type_name and ?type_name (i.e. same type_name)

2. ?type_name and !value_id (i.e. value_id is of sort type_name)

3. !value_expression and !value_expression
(i.e. the same value_expression)

In the specification given below the LOTOS type of synchronisation convention is assumed for the indices of actions. This implies that synchronisable indexed actions are in a communication relation with each other.

Requirement descriptions are obtained from the standard for the OSI connection oriented transport protocol - [41]. It is assumed that simple canonical transformations of the behaviour mechanisms described in the standard yield partial constraints (as in the Partial Constraint 6.1). For example, the procedure describing Connection establishment (cf.clause 6.5.4 of [41]) contains the following statement:

"A transport connection is established by means of one transport entity (the initiator) transmitting a CR TPDU to the other transport entity (the responder), which replies with a CC TPDU."

Where CR TPDU and CC TPDU are a connect request and connect confirm transport protocol data units respectively.

By assigning roles of initiator/responder to both the gates p1 and p2 we deduce the following partial constraint on the actions of p1 and p2:

"p1(send/output) CR TPDU precedes p2(receive/input) CR TPDU
precedes p2(send/output) CC TPDU
precedes p1(receive/input) CC TPDU"

or

"p1(send/output) CR TPDU precedes p1(receive/input) CC TPDU

In a similar manner we obtain partial constraints for PE1 and PE2 as follows:

PE1 partial constraints:

" 't1?TConreq precedes p1!NDatareq!Conreq'

or

't1?TConresp precedes p1!NDatareq!Conconf'

or

'p1?NDataind?Conreq precedes t1!TConind'

or

'p1?NDataind?Conind precedes t1!TConconf' " (6.2)

PE2 partial constraints:

" 't2?TConreq precedes p2!NDatareq!Conreq'

or

't2?TConresp precedes p2!NDatareq!Conind '

or

'p2?NDataind?Conreq precedes t2!TConind'

or

'p2?NDataind?Conind precedes t2!TConconf' (6.3)

Using a LOTOS-like notation for the specification of processes satisfying the partial constraints we get:

protocol[t1,p1,t2,p2] = PE1[t1,p1,t2,p2] || PE2[t1,p1,t2,p2]

(6.4)

where

PE1[t1,p1,t2,p2] =

t1?TConreq ; p1!NDatareq!Conreq ; PE1[t1,p1,t2,p2]

[]

t1?TConresp ; p1!NDatareq!Conind ; PE1[t1,p1,t2,p2]

[]

p1?NDataind?Conreq ; t1!TConind ; PE1[t1,p1,t2,p2]

[]

p1?NDataind?Conind ; t1!TConconf ; PE1[t1,p1,t2,p2]

PE2[t1,p1,t2,p2] =

t2?TConreq ; p2!NDatareq!Conreq ; PE2[t1,p1,t2,p2]

[]

t2?TConresp ; p2!NDatareq!Conind ; PE2[t1,p1,t2,p2]

[]

p2?NDataind?Conreq ; t2!TConind ; PE2[t1,p1,t2,p2]

[]

p2?NDataind?Conind ; t2!TConconf ; PE2[t1,p1,t2,p2]

The partial behaviour of the underlying service processes obtained from its partial constraints is given by:

unds[p1,p2] = unds1[p1,p2] || unds[p1,p2]

p1?NDatareq?Conreq ; p2!Ndataind!Conreq ; unds1[p1,p2]

[]

p1?NDatareq?Conind ; p2!Ndataind!Conind ; unds1[p1,p2]

||

p2?NDatareq?Conreq ; p1!Ndataind!Conreq ; unds2[p1,p2]

[]

p2?NDatareq?Conind; p1!Ndataind!Conind ; unds2[p1,p2]

(6.5)

The partial behaviour of the user processes must satisfy the intended service. The user processes are given by:

usr[t1,t2] = user[t1,t2] || user2[t1,t2] (6.6)

where

user1[t1,t2] = t1!TConreq ; t1?TConconf ; user1[t1,t2]

[]

t1?TConind ; t1?TConresp ; user1[t1,t2]

user2[t1,t2] = t2!TConreq ; t2?TConconf ; user2[t1,t2]

[]

t2?TConind ; t2?TConresp ; user2[t1,t2]

The system S is obtained from the parallel composition of all the defined processes. Parallel composition is the same as in LOTOS, so that stepwise synchronisation of matching events of two processes occurs, at common interaction points. For example, two actions at the interaction point 'p1' can synchronise with exchange of messages if the two processes offer: '!NDatareq!Conreq' and '?NDatareq?Conreq' respectively. Where '?Conreq' for example is an index representing a predefined sort of a pdu (protocol data unit) type and '!Conreq' an actual value in a manner similar to that of ACTONE data types.

However, note that since deadlock occurs in a system only when all processes are blocked forever, the parallel composition of all the partially specified processes must allow for all synchronised interactions which can occur, to indeed occur. Other processes with event offers wait for synchronisation until a peer becomes ready to synchronise.

The partial specification of Equations (6.2) and (6.3) allow for true concurrency between connections, i.e. both sides may initiate connect requests concurrently, however the specification is transparent to crossovers.

[end of example 6.1]

6.3 Validation Method

The composed distributed system is valid if it is syntactically and semantically consistent. Syntactic consistency can be checked either manually or automatically. Syntactic consistency is demonstrated by paralle' composition of the protocol process with the underlying service. If the composition results in deadlock free observable traces then the system is syntactically valid. Semantic consistency is demonstrated by parallel composition of the protocol-underlying service with user processes in the environment. If such a composition results in deadlock-free behaviour traces, then consistency with intended service is validated.

If formal specifications of the service (or user processes) is not available, then parallel composition of the protocol process and the underlying service yields traces. Partial constraints are constructed from the observed traces, from which behaviour expressions satisfying them are obtained. Partial constraints satisfying the behaviour expressions are compared to partial constraints obtained from service descriptions for consistency.

As an illustration of the validation method, consider the following:

Example 6.2

Parallel composition of the processes of Example 6.1 (see also Figures 6.1 and 3.1) gives the expression:

$$S = usr[t1,t2] \;|[t1,t2]|\; protocol[t1,p1,t2,p2] \;|[p1,p2]|\; unds[p1,p2] \qquad (6.7)$$

Common actions with matching events of the two processes result in interactions. Interactions are possible at each recursion of the processes. Those actions not common to any of the processes are not synchronised, but interleaved. The composition results in traces which yield partial constraints consistent with those traces. Behaviour expressions satisfying the partial constraints represent possible behaviours of the system. The set of observable traces for the behaviour specified in Equation (6.7) are:

<
t1!TConreq ,
p1!Ndatareq!Conreq ,
p2!NDataind!Conreq ,
t2!TConind ,
t2!TConresp ,
p2!NDatareq!Conind ,
p1!NDataind!Conind ,
t1!TConconf
or
t2!TConreq ,
p2!NDatareq!Conreq ,
p1!NDataind!Conreq ,
t1!TConind ,
t1!TConresp ,
p1!NDatareq!Conind ,
p2!NDatareq!Conind ,
t2!TConresp > (6.8)

In a given system of determination of partial constraints from traces, determinations of actions can be made upto a predefined depth of determination. The set of traces (6.8) represent possible behaviours of the connection service.

The trace (6.8) forms the basis of determining the constraint:

" 't1!TConreq precedes
p1!Ndatareq!Conreq precedes
p2!NDataind!Conreq precedes
t2!TConind precedes
t2!TConresp precedes
p2!NDatareq!Conind precedes
p1!NDataind!Conind precedes
t1!TConconf'
or
't2!TConreq precedes
p2!NDatareq!Conreq precedes
p1!NDataind!Conreq precedes
t1!TConind precedes
t1!TConresp precedes
p1!NDatareq!Conind precedes
p2!NDatareq!Conir ! precedes
t2!TConresp' " (6.9)

The constraint (6.9) can be compared to the time sequence diagrams of the OSI Transport service definition standard (cf. pps. 9, Figure 4 [42]).

For example, the time sequence diagrams of the transport service contain the sequence of transport service primitives given in Figure 6.2:

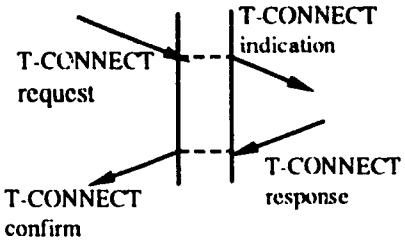


Figure 6.2 Successful TC establishment

The following partial constraint based on the time sequence diagram of Figure 6.2 describes successful TC establishment when TS user1 (interacting with TS provider at interaction point t1) or TS user2 (interacting with TS provider at interaction point t2) is calling/called TS user:

" (t1/t2)!TCONreq precedes (t1/t2)!TCONconf " (6.10)

It is easy to deduce the partial constraint (6.10) from (6.9) and trace (6.8), since (6.10) is consistent with (6.9). Therefore, the protocol as specified in Equation (6.4) is valid with respect to intended service provided to the user as specified in Equation (6.6).

[end of Example 6.2]

Example 6.3

The previous example can be easily extended to include the service consisting of rejection of TC establishment requests by a called TS user or TS provider (see also Figures 6.1 and 3.1).

Disconnect Requests And Indications

The additional intended service to the user is expressed by the following partial constraint on user processes:

" '(t1/t2) T_Conreq precedes (T_Conconf *or* T_DISind)'

or

'(t1/t2)T_Conind precedes (T_Conresp *or* T_DISreq)' " (6.11)

So that the user processes of Equation (6.6) are modified as follows:

usr[t1,t2] = user1[t1,t2] || user2[t1,t2] (6.12)

where

user1[t1,t2] = t1!TConreq ; (t1?TConconf [] t1?TDISind) ; user1[t1,t2]

[]

t1?TConind ; (t1?TConresp [] t1?TDISreq); user1[t1,t2]

user2[t1,t2] = t2!TConreq ; (t2?TConconf [] t2?TDISind) ; user2[t1,t2]

[]

t2?TConind ; (t2?TConresp [] t2?TDISreq); user2[t1,t2]

and the partial constraints for PE1, PE2 and the underlying service are modified as follows:

PE1:

" 't1?TConreq precedes p1!NDatareq!Conreq'

or

't1?TConresp precedes p1!NDatareq!Conind'

or

't1?TDISreq precedes p1!NDatareq!TDISreq'

or

'p1?NDataind?Conreq precedes t1!TConind'

or

'p1?NDataind?TDISreq precedes t1!TDISind'

or

'p1?NDataind?Conind precedes t1!TConconf' " (6.13)

PE2:

" 't2?TConreq precedes p2!NDatareq!Conreq'

or

't2?TConresp precedes p2!NDatareq!Conind'

or

't2?TDISreq precedes p2!NDatareq!TDISreq'

or

'p2?NDataind?Conreq precedes t2!TConind'

or

'p2?NDataind?TDISreq precedes t2!TDISind'

or

'p2?NDataind?Conind precedes t2!TConconf' " (6.14)

Accordingly, the specification in Equation 6.4 is modified as follows:

protocol[t1,p1,t2,p2] = PE1[t1,p1,t2,p2] || PE2[t1,p1,t2,p2] (6.15)

where

PE1[t1,p1,t2,p2] =

t1?TConreq ; p1!NDatareq!Conreq ;

PE1[t1,p1,t2,p2]

```
        []
         t1?TConresp ; p1!NDatareq!Conconf ;
                                                    PE1[t1,p1,t2,p2]
        []
         t1?TDISreq ; p1!NDatareq!TDISreq ;
                                                    PE1[t1,p1,t2,p2]
        []
         p1?NDataind?Conreq ; t1!TConind ;
                                                    PE1[t1,p1,t2,p2]
        []
         p1?NDataind?TDISreq ; t1!TDISind ;
                                                    PE1[t1,p1,t2,p2]
        []
         p1?NDataind?Conind ; t1!TConconf ;
                                                    PE1[t1,p1,t2,p2]
PE2[t1,p1,t2,p2] =
         t2?TConreq ; p2!NDatareq!Conreq ;
                                                    PE2[t1,p1,t2,p2]
        []
         t2?TConresp ; p2!NDatareq!Conconf ;
                                                    PE2[t1,p1,t2,p2]
        []
         t2?TDISreq ; p2!NDatareq!TDISreq ;
                                                    PE2[t1,p1,t2,p2]
```

[]

p2?NDataind?Conreq ; t2!TConind ;

PE2[t1,p1,t2,p2]

[]

p2?NDataind?TDISreq ; t2!TDISind ;

PE2[t1,p1,t2,p2]

[]

p2?NDataind?Conind ; t1!TConconf ;

PE2[t1,p1,t2,p2]

the underlying service is also appropriately modified:

unds[p1,p2] = unds1[p1,p2] || unds2[p1,p2]

p1?NDatareq?Conreq ; p2!Ndataind!Conreq ; unds1[p1,p2]

[]

p1?NDatareq?TDISreq ; p2!Ndataind!TDISreq ; unds1[p1,p2]

[]

p1?NDatareq?Conind ; p2!Ndataind!Conind ; unds1[p1,p2]

||

p2?NDatareq?Conreq ; p1!Ndataind!Conreq ; unds2[p1,p2]

[]

p2?NDatareq?TDISreq ; p1!Ndataind!TDISreq ; unds2[p1,p2]

[]

p2?NDatareq?Conind ; p1!Ndataind!Conind ; unds2[p1,p2]

(6.16)

Once again the intended service can be validated by means of Equation 6.7 and its observable traces.

Collision Detection

Collision of connect requests can occur if PE1 or PE2 receives a connect request from the other side before or after it sends a connect request via the underlying service. This can occur only if the underlying service delivers a connect request before or after either side sends a connect reques'. In order to detect and avoid this collision, both entities non-deterministically ignore the incoming request and proceed to establish the connection for the connect request it has accepted from its own user. To implement this, the protocol entities as specified in Equation 6.14 are modified as follows:

```
PE1[t1,p1,t2,p2] =
    t1?TConreq ; (p1!NDatareq!Conreq ;
                        (p1?NDataind?Conreq' ; protocol[t1,p1,t2,p2]
                         []
                          PE1[t1,p1,t2,p2])
                  []
                   p1?NDataind?Conreq' ; p1!NDatareq!Conreq ;
                                          PE1[t1,p1,t2,p2]
    []
    t1?TConresp ; p1!NDatareq!Conconf ; PE1[t1,p1,t2,p2]
    []
    p1?NDataind?Conreq ; t1!TConind ; PE1[t1,p1,t2,p2]
    []
    p1?NDataind?Conind ; t1!TConconf ; PE1[t1,p1,t2,p2]
```

with PE2 defined similarly. (6.17)

A possible deadlock or race condition ensuing from both sides respectively ignoring repeatedly the other sides request is resolved by a 'fair' environment assumed in Assumption 5.4.1. In a fair environment one side will eventually get through, when the other side stops sending connect requests.

Data transfer is now introduced in the example:

Data Transfer

The data transfer phase of the protocol is always preceded by the connection establishment phase, therefore we specify the protocol including the data transfer phase as follows:

protocol[t1,p1,t2,p2] =PE1[t1,p1,t2,p2] || PE2[t1,p1,t2,p2] (6.18)

where

PE1[t1,p1,t2,p2] = conPE1[t1,p1,t2,p2] [] dtrfrPE1[t1,p1,t2,p2] ;

PE2[t1,p1,t2,p2] = conPE2[t1,p1,t2,p2] [] dtrfrPE2[t1,p1,t2,p2] ;

Where PE1 and PE2 are modified to obtain conPE1 and conPE2 respectively as follows:

```
conPE1[t1,p1,t2,p2] =
        t1?TConreq ; p1!NDatareq!Conreq ;
                                        conPE1[t1,p1,t2,p2]
        []
        t1?TConresp ; p1!NDatareq!Conconf ;
                                        conPE1[t1,p1,t2,p2]
        []
        t1?TDISreq ; p1!NDatareq!TDISreq ;
                                        conPE1[t1,p1,t2,p2]
```

```
        []
        p1?NDataind?Conreq ; t1!TConind ;
                                                conPE1[t1,p1,t2,p2]
        []
        p1?NDataind?TDISreq ; t1!TDISind ;
                                                conPE1[t1,p1,t2,p2]
        []
        p1?NDataind?Conind ; t1!TConconf ;
                                                dtrfrPE1t1,p1,t2,p2]

conPE2[t1,p1,t2,p2] =
        t2?TConreq ; p2!NDatareq!Conreq ;
                                                conPE2[t1,p1,t2,p2]
        []
        t2?TConresp ; p2!NDatareq!Conconf ;
                                                conPE2[t1,p1,t2,p2]
        []
        t2?TDISreq ; p2!NDatareq!TDISreq ;
                                                conPE2[t1,p1,t2,p2]
        []
        p2?NDataind?Conreq ; t2!TConind ;
                                                conPE2[t1,p1,t2,p2]
        []
        p2?NDataind?TDISreq ; t2!TDISind ;
                                                conPE2[t1,p1,t2,p2]
```

[]

p2?NDataind?Conconf ; t2!TConconf ;

dtrfrPE2[t1,p1,t2,p2]

(6.19)

And the data transfer process is defined as:

where

dtrfrPE1[t1,p1,t2,p2] = t1?TDatareq ; p1!NDatareq!Datareq ;

dtrfrPE1[t1,p1,t2,p2]

[]

p1!NDataind!Datareq ; t1!TDataind ;

dtrfrPE1[t1,p1,t2,p2]

[]

p1!NDatareq!s_AK ; dtrfrPE1[t1,p1,t2,p2]

[]

p1?NDataind?r_AK ; dtrfrPE1[t1,p1,t2,p2]

with dtrfrPE2 similarly defined on t2 and p2.

The underlying-service can be similarly modified to consist of two processes such that the data transfer process component is defined as:

dtrfrunds[p1,p2] = dtrfrunds1[p1,p2] || dtrfrunds2[p1,p2]

dtrfrunds1[p1,p2] = p1?NDatareq?Datareq ; p2!NDataind!Datareq ;

dtrfrunds1[p1,p2]

[]

p1?NDatareq?s_AK ; p2!NDataind!r_AK

dtrfrunds1[p1,p2]

||

p2?NDatareq?Datareq ; p1!NDataind!Datareq ;

dtrfrunds2[p1,p2]

[]

p2?NDatareq?s_AK ; p1!NDataind!r_AK

dtrfrunds2[p1,p2]

(6.20)

The user process definitions are analogous. Once again the composition of processes as in Equation 6.7 can be shown to satisfy the intended service, in a manner similar to the above.

Expedited Data

However, the transport service standard imposes special constraints on data transfer. The special constraints are that of re-ordering and deletion. We will consider re-ordering.

Expedited TSDUs are expected to be delivered ahead of any normal data which is already in the service queue, and is in the process of being transmitted (cf. clause 9.2 of [IS2]). This can be expressed by the appropriate partial constraints and the corresponding process dtrfrPE1 is defined as:

```
dtrfrPE1[t1,p1,t2,p2] = t1?TDatareq ;
                        ((t1?TExpdatareq ;
                          p1!NDatareq!Expdatareq ;
                          p1!NDatareq!Datareq)
                          []
                          (t1?TDatareq' ;
                            p1!NDatareq!Datareq ;
```

p1!NDatareq!Datareq')) ;

dtrfrPE1[t1,p1,t2,p2]

[]

p1!NDataind!Datareq ; t1!TDataind ;

dtrfrPE1[t1,p1,t2,p2]

[]

p1!NDataind!Expdatareq ; t1!TExpdataind ;

dtrfrPE1[t1,p1,t2,p2]

with dtrfrPE2 similarly defined...

(6.21)

Note the primes on the value expression indexing the actions imply a value different from the previous one of the same sort.

The service processes are appropriately modified as follows:

dtrfrunds[p1,p2] =

p1?NDatareq?Datareq ;

(p1?NDatareq?Expdatareq ;

p2!NDataind!Expdatareq ; p2!NDataind!Datareq ; dtrfrunds1[p1,p2]

[]

p1?NDatareq?Datareq' ;

p2!NDataind!Datareq ; p2!NDataind!Datareq' ; dtrfrunds1[p1,p2])

||

p2?NDatareq?Datareq ;

(p2?NDatareq?Expdatareq ;

p1!NDataind!Expdatareq ; p1!NDataind!Datareq ; dtrfrunds2[p1,p2]

[]

p2?NDatareq?Datareq' ;

p1!NDataind!Datareq ; p1!NDataind!Datareq' ; dtrfrunds2[p1,p2])

(6.22)

And the user processes:

usr[t1,t2] = user1[t1,t2] || user2[t1,t2] (6.23)

where

user1[t1,t2] = conuser1[t1,t2] [] dtrfruser1[t1,t2] ;

user2[t1,t2] = conuser2[t1,t2] [] dtrfruser2[t1,t2]

and

dtrfruser1[t1,t2] = t1!TDatareq ; dtrfruser1[t1,t2]

[]

t1!TExpdatareq ; dtrfruser1[t1,t2]

with dtrfruser2[t1,t2] similarly defined.

Flow Control By Backpressure

Internal conditions of the receiving user or the service during the data transfer phase or of the provider may cause flow control to be exercised by the transport protocol. Flow control results in the user being unable to add normal data to the service queue when that would prevent addition of an expedited TSDU (Transport service data unit) (cf. clause 9.2 [42]).

The adding of normal data to the queue, is preceded by an action denoting flow-control-ready in the process dtrfrPE2. The flow-control-not-ready condition is interpreted as representing the condition: the receiving end user is not ready to accept another NDatareq. The flow control not ready condition is also true when the service queue is full but for one more data element (which must be reserved for expedited data). We denote the flow control conditions by the following actions:

que_full:

denotes service provider is ready to receive only one more element (expedited data)

que_ not_full:

denotes service provider is ready to receive data requests

usr_rdy:

denotes receiving protocol entity ready to receive data

usr_not_rdy:

denotes receiving protocol entity not ready to receive data but for an expedited data

flow_control_rdy:

denotes the receiving user is ready to accept data from the service

flow_control_not_rdy:

denotes the receiving user is not ready to accept data from the service, but for one expedited element

For the sake of simplicity, if we assume that only user1 initiates TDatareqs, the process dtrfrPE2 is modified as follows:

```
dtrfrPE1[t1,p1,t2,p2] =
                        que_not_full ;
                        (t1?TDatareq ;
                          ((t1?TExpdatareq ;
                            p1!NDatareq!Expdatareq ;
                            p1!NDatareq!Datareq)
                            []
                            (t1?TDatareq' ;
                             p1!NDatareq!Datareq ;
                              p1!NDatareq!Datareq')) ;
                                    dtrfrPE1[t1,p1,t2,p2])
                       []
                        que_full ; (dtfrPE1[t1,p1,t2,p2]
                                []
                                t1?TExpdata ; p1!Ndatareq!Expdata ;
                                        dtrfrPE1[t1,p1,t2,p2])
```

(6.24)

The under-lying-service is modified as follows:

```
dtrfrunds[p1,p2] =
usr_rdy ; que_not_full ;
(p1?NDatareq?Datareq ;
    (p1?NDatareq?Expdatareq ;
```

```
        p2!NDataind!Expdatareq ; p2!NDataind!Datareq ; dtrfrunds[p1,p2]
      []
      p1?NDatareq?Datareq' ;
       p2!NDataind!Datareq ; p2!NDataind!Datareq' ; dtrfrunds[p1,p2])
[]
p2?NDatareq?Datareq ;
    (p2?NDatareq?Expdatareq ;
       p1!NDataind!Expdatareq ; p1!NDataind!Datareq ; dtrfrunds[p1,p2]
      []
       p2?NDatareq?Datareq' ;
       p1!NDataind!Datareq ; p1!NDataind!Datareq' ; dtrfrunds[p1,p2])
[]
usr_not_rdy ; que_full ;
      (dtrfrunds[p1,p2]
      []
      p1?NDatareq?Expdatareq ; p2!NDataind!Expdatareq ;
                                          dtrfrunds[p1,p2])
```

(6.25)

The receiving entity dtrfrPE2 is modified as follows:

```
dtrfrPE2[t1,p1,t2,p2] =  flow_control_rdy ; usr_rdy ;
                              (p2?NDataind?Datareq ; t1!TDataind
                               []
                               p2?NExpdataind?Expdataind ;
                                  t1!TExpdataind) ; dtrfrPE2[t1,p1,t2,p2]
```

[]

flow_control_not_rdy ; usr_not_rdy ;

(p2? NExpdataind?Expdataind ;

t2!TExpdataind ; dtrfrPE2

[]

dtrfrPE2) (6.26)

The user process user2 is also modified as follows:

dtrfruser2[t1,t2] = flow_control_rdy ;

(t2?TDataind [] t2?TExpdataind) ; dtrfruser1[t1,t2]

[]

flow_control_not_rdy ; t1?TExpdataind ;

dtrfruser2[t1,t2]

(6.27)

The Equation 6.7 is modified as follows:

S = usr[t1,t2] |[t1,t2]| protocol[t1,p1,t2,p2]

|[p1,p2]| unds[p1,p2] (6.28)

The observable traces of Equation (6.27)` yield behaviour expressions which satisfy the service constraints with respect to flow control.

[end of example 6.3]

6.4 A Proof Technique For Verification

The proof technique used for verification of logical properties of specifications is based on Assumption (5.4.1) of a 'fair' environment and the method of Invariants [36],[21]. It is briefly described next.

6.4.1 Proof Technique

Assumption 6.1 Partial Specifications

Process algebraic specifications obtained from partial constraints generate partial specifications in the form of guarded process algebraic equations with no hidden actions.

[end of Assumption 6.1]

The class of partial specifications treated by the proof technique described below is the class of process algebraic guarded recursive equations which generate Finite State Transition (FST) systems (see also Chapter 3 Definition 3.1.1).

Process algebraic guarded recursive equations which can generate FSTs can be shown to have a finite (or countable) set of infinitely recurring traces. The partial constraint proof technique is used to verify logical properties of system behaviours on the basis of finite (or countable) sets of infinitely recurring traces.

Proof Method

The proof technique is based on inference rules for partial specifications given below:

A guarded recursive process algebraic equation has the form:

$P[\alpha,\beta,...] = \alpha.\beta. \ldots .P [] \gamma.\delta. \ldots .P [] \ldots$ (6.29)

where $\alpha,\beta,\gamma,\delta$ are actions which belong to the alphabet of the process. The process P may have sub processes with a similar structure.

In a 'fair' communication environment E:

Let pc,pcI1,pcI2...pcO1,pcO2,... denote well formed partial constraints (Definition 4.4.2) on the actions of E.

Let $\{\alpha,\beta,\gamma,...\} \in E$

Let P,P1,P2... be guarded recursive processes defined on the actions of E satisfying partial constraints on behaviours in E. And for any process P, let P---->P' denote the transition of P to P' with the occurrence of an action.

Then, the following inference rules are used for verification of the processes of E:

1. If $P = \alpha.P$ then

If the partial constraint pcI = 'α' true before instantiation of P

then

pcI P |= pcO is true

where pcO = pcI (I) is true after every recursive instantiation of P, and

P |= the trace $<\alpha,\alpha,\alpha,...>$.

Note that the trace $<\alpha,\alpha,\alpha,...>$ ==> pcO at depth of determination 1. (6.30)

2. (i) If P = $\alpha.\beta.P$ then

If the partial constraint pcI = 'α precedes β' is true before instantiation of P

then

If P --α--> β.P ==>

pcI 'P --α--> β.P' |= 'α' is true now and pcO will *eventually* be true

where pcO = pcI (I) is true after every recursive instantiation of P, and

P |= the trace

<$\alpha\beta$,$\alpha\beta$,...$\alpha\beta$,... etc.>

Note that the traces imply pcO at depth of determination 2. (6.31)

2. (ii) If P = $\alpha.\beta.P$ then

If the partial constraint pcI = 'α precedes β' is true before instantiation of P

then

pcI 'P --β--> P' |= pcO is true now

where pcO = pcI (I) is true after every recursive instantiation of P, and

P |= the trace

<$\alpha\beta$,$\alpha\beta$,...$\alpha\beta$,... etc.> (6.32)

Note that the traces imply pcO at depth of determination 2.

3. If P = (α || β).P then

If the partial constraint pcI = 'α or β' is true before instantiation of P

then

pcI 'P--α-->P' |= 'α 'is true now and pcO will *eventually* be true

where pcO = pcI (I) is true after every recursive instantiation of P, and

P |= the trace

<αα,αα,...ββ,... or

ββ,ββ,...αα,... or

αα,αα,...αβ,... or

αα,αα,...βα,... or

ββ,ββ,...αβ,... or

ββ,ββ,...βα,... etc. >.

Note 1: that the traces imply pcO at depth of determination 1 and 2.

Note 2: A similar inference rule holds if 'P--β-->P' (6.33)

4. Sequence rule:

If

pcI1'P1---->P2' |= pcO1 and pcO1 'P2---->' |= pcO2 then

'pcI1precedes pcO2' is true for P1 (6.34)

5. Choice rule:

If

pcI1'P1' |= pcO1 or pcI2 'P2' |= pcO2 then

'pcI1 *or* pcI2' 'P1|| P2' |= 'pcO1 *or* pcO2' (6.35)

6. Conjunction rule:

If

pcI1'P1' |= pcO1 and pcI2 'P2' |= pcO2 then

'pcI1*and* pcI2' 'P1||P2' |= 'pcO1 *and* pcO2' (6.36)

7. Now let 'proc' be a behaviour process, and pcI a partial constraint defined in E such that pcI is true at the instantiation of 'proc'.

Let 'Spec' be the set of processes in the environment of 'proc';

Let process 'proc' be parallel composed with all the other processes of its environment E in accordance with the Formulae (5.54) of Chapter 5. The parallel composition of the processes is assumed to instantiate the processes. Then if the parallel composition is denoted by Parallel(Spec), and:

pcI Parallel(Spec) |= trO ==> pcO' (by determinations of depths 1 and 2)

and

beO be s.t. beO |= pcO' ==> pcO (6.37)

where the partial constraint pcI is true at instantiation of the processes of Spec. The trace set trO is the set of all observable traces derived from Parallel(Spec) which can eventually be observed. The set of traces trO forms the basis of determinations at depth 1 and 2 of the partial constraints pcO'. The partial constraints pcO' turn out to be finite sequences of actions in trO mapped onto precedence relations of actions(at a given depth of determination) to form guarded recursive sub-processes. The sub-processes so obtained for the traces in trO are related by the *'or'* relation. The behaviour expression beO is constructed s.t. beO |= pcO' (see also Chapters 4 and 5). The construction of beO from trO is also illustrated in the validation examples above.

If the partial constraints pcO obtained as a result of applying the inference rules (6.30-6.37)) are consistent with the logical property to be verified, then the process 'proc' is said to be verified with respect to that property.

Deadlock

Note that fairness of an environment E implies that, E is an operational environment which does not deadlock on any of the actions of the processes specified in Parallel(Spec) due to an action not in Parallel(Spec).

However, deadlock can occur if an action in Parallel(Spec) cannot occur now or eventually. In such a case parallel spec satisfies the empty (' ') constraint.

For example the parallel(spec):

'α precedes β' $\alpha.\beta|\beta.\alpha \models$ deadlock ==> ' ' constraint from rule (6.32)

(where α and β do not communicate)

Example 6.4

Consider the CCS partial specifications:

$S1 = \alpha$ and $S2 = \alpha$ (6.38)

and

'α' $S = S1|S2 \models \langle\alpha\rangle \Longrightarrow$ 'α' from the inference rules (6.31) and (6.36)

(6.39)

In (6.38), S1,S2 satisfy the constraint "α" in a fair environment E, since "α" is always true. Hence, the behaviours S1 and S2 are safe with respect to the property "α". The expression (6.39) shows that S satisfies the liveness property 'α'.

That means for the behaviour S, 'α' will eventually be true. But the safe operation of S cannot be asserted from (6.39). However, since 'α' was true before instantiation of S, and 'α' is eventually true on termination of S. Therefore, S = α.S (as constructed from the trace <α> and rule (6.37), is also safe.

Verification of the property safety property 'α' can be proved inductively:

If the partial constraint 'α' is true before the instantiation of S, then it is true when S terminates successfully. i.e. S is verified to be safe w.r.t. the property 'α'.

Since:

'α' α|α |= <α> (the trace α is observable) (by rule 6.37)

==> α (the behaviour α) (by rule 6.37)

==> 'α' is true (the partial constraint 'α' is true)

(by rules 6.36 and 6.37)

therefore: S is verified w.r.t. the safety property 'α'

And

For S=S1|S2:

'α' S1|S2 |= <α> (the trace α of S is eventually observable)

==> 'α' is eventually true (by rule 6.36,6.37)

(the partial constraint 'α' is eventually true)

therefore: S is verified w.r.t. the liveness property 'α'.

If S1 and S2 are recursively defined to be:

S1 = α.S1 and S2 = α.S2 (6.40)

and

$S = S1|S2 \models \langle\alpha,\alpha,\alpha,\alpha,\alpha....\rangle$ (6.41)

==> 'α' eventually true (for every finite prefix) by rules (6.36,6.37)

==> 'α' always true (for every finite prefix)

(since 'α' is true at instantiation of S and when it terminates)

therefore: S is verified w.r.t. the safety property 'α'.

In (6.41) S satisfies 'α', since for every finite prefix of the sequence of αs, 'α' is true. Therefore, 'α' is always true. Hence, the behaviour S is safe with respect to the property 'α'.

[end of example 6.4]

Example 6.5

The partial constraint 'α *or* β' is a 'weak' constraint satisfied by the behaviour S:

$S = (\alpha + \beta).S1 \| (\alpha + \beta).S2$ (6.42)

where

$S1 = (\alpha + \beta).S1$

$S2 = (\alpha + \beta).S2$

To prove that S behaves safely w.r.t. the property 'α *or* β', we write the expression:

'α *or* β' $(\alpha + \beta).S1 \| (\alpha + \beta).S2$

$\models \langle\alpha\alpha,\alpha\alpha,...\beta\beta,...$ or

$\beta\beta,\beta\beta,...\alpha\alpha,...$ or

$\alpha\alpha,\alpha\alpha,...\alpha\beta,...$ or

$\alpha\alpha,\alpha\alpha,...\beta\alpha,...$ or

$\beta\beta,\beta\beta,...\alpha\beta,...$ or

$\beta\beta,\beta\beta,...\beta\alpha,...$ etc. $\rangle$

(by rules 6.33,6.35,6.36 and 6.37)

$\models \Diamond$'α *or* β' (is eventually true)

$\Longrightarrow$ $\Box$'α *or* β' (is always true)

since it is true before instantiation and after each recursion

$\Longrightarrow (\alpha + \beta).S$ (6.43)

therefore: $S = (\alpha + \beta).S \models \Box$'$\alpha$ *or* β', and S is safe w.r.t. the property 'α *or* β'

Also, since:

'α *or* β' $(\alpha + \beta).S1 \| (\alpha + \beta).S2$

$\models <(\alpha + \beta).(\alpha + \beta),(\alpha + \beta),(\alpha+\beta),...>$

$\models <(\alpha\alpha + \beta\beta + \alpha\beta + \beta\alpha), (\alpha + \beta)...>$

$\Longrightarrow <\alpha\alpha...$ or $\beta\beta...$ or $\alpha\beta...$ or $\beta\alpha...$

(possible behaviours traces>

(by rules 6.32,6.35,6.36 and 6.37)

$\Longrightarrow$ 'α precedes α... *or*

'β precedes β... *or*

'α precedes β... *or*

'β precedes α... (by rule 6.32) (6.44)

Therefore, if α occurs now, then β will eventually occur and vice-versa. Therefore, S is live w.r.t. the 'strong' property "'α precedes β' *or* 'β precedes α'". But S is not safe w.r.t. this property since it is not true for every finite prefix of the traces of S.

[end of example 6.5]

6.5 Verification Of The Alternating Bit Protocol

The method of verification is illustrated by means of the Alternating Bit protocol specified in Chapter 5.

The protocol is verified for safety and liveness properties. Safety is proved by showing that safety invariants transformed into partial constraints are always true throughout the operation of the protocol. i.e S |= safety constraint pcI ==> pcI is always true. Liveness is proved by showing that a given property will eventually be true i.e S satisfies the pc before instantiation, and S |= liveness constraint pcI, pcI will eventually be true after instantiation.

For ease of comprehension, the denotational meaning of the actions of the AB protocol is reproduced below (from Chapter 5):

For user-protocol interactions:

Let su,ru denote the 'Send' action and 'Receive' action by users.

For protocol-medium interactions:

Let s0 and s1 denote the sending of messages with sequence number 0 and 1 respectively and let r0 and r1 denote corresponding receive actions.

For acknowledgements, let sack0 and sack1 represent send actions and rack0 and rack1 corresponding receives.

For the timer let tout0, tout1, tstart0 and tstart1 represent timeout and timestart actions for messages with sequence number 0 and 1 respectively.

Safety

Safety properties for the sender process (S1), the receiver process (S2) and the medium are stated. These properties are converted into partial constraints by simple canonical transformations (the transformation rules are the designers choice) as follows:

A safety invariant of the sender process (S1) is:

Safety Invariant 6.1

"A message is not sent until an acknowledgement has been received for the previous message"

Partial Constraint

" not ('s0 precedes rack1') "

and

" not ('s1 precedes rack0') " (6.45)

For the receiver process (S2) is:

Safety Invariant 6.2

" A message is acknowledged only after it has been received"

" 'r0 precedes sack0'"

and

" 'r1 precedes sack1' " (6.46)

For the medium is:

Safety Invariant 6.3

"The output sequence of messages is at all times atmost one shorter than the input sequence" (6.47)

Definition 6.1 Message Extraction Function

Define m to be a function which extracts a message sequence from a partial constraint as follows:

m(s0/s1) = ms0/ms1 and m(r0/r1) = mr0/mr1

so that:

m('s0 precedes r0 precedes s1')

==> ms0,mr0,ms1`.

==> minput (number of input messages) = 2 and

moutput (number of output messages) = 1

and

m('s0 precedes r0 precedes s1 precedes r1 precedes s0')

==> ms0,mr0,ms1,mr1,ms0'

==> minput ≤ moutput with minput = moutput +1 for all

values of minput,moutput (6.48)

where s0,s1 can be considered to be input and r0/r1 as output actions respectively. The primes distinguish different messages with the same message sequence number. And if an input action follows an output action it is assumed that the message is a new one, different from the one output in the preceding output action.

[end of Definition 6.1]

Therefore, in order to prove the property stated in the Safety Invariant 6.3, we have to prove the partial constraint associated with the safety property.

Partial Constraint

" 's0 precedes r0 precedes s1' " is always true (6.49)

and a similar property with s1 as the first action.

The safety property for the system is:

Safety Invariant 6.4

"The output sequence of messages is at all times an initial sub-sequence of the input sequence" (6.50)

The function m when applied to the constraint:

" 'us precedes ur'" ==> msu,mru,msu,mru,msu....

where each pair of su's preceding ru refer to the same message.

6.5.1 Verification Proofs

1. Proof Of Safety Invariant 6.1

"not ('s0 precedes rack1')"

and

"not('s1 precedes rack0')" (6.51)

Proof:

Consider the observable traces of the service, derived from 5.82 and construct the partial constraint:

" 'rack1 precedes su precedes s0 precedes tstart0 precedes

('tout0 precedes s0 precedes tstart0 precedes

tout0 precedes s0 precedes tstart0 precedes

...'

or 'r0 precedes ru precedes sack0 precedes pc11' ')'

or

'rack0 precedes su precedes s1 precedes tstart1 precedes

('tout1 precedes s1 precedes tstart1 precedes

tout1 precedes s1 precedes tstart1 precedes

...'

or 'r1 precedes ru precedes sack1 precedes pc10' ')' "

(6.52)

The behaviour constructed in (5.84) of chapter 5 satisfies the partial Constraint (6.52):

S = rack0.su.s1.tstart1.(tout1.s1.tstart1.L1 + r1.ru.sack1.S10')

+

rack1.su.s0.tstart0.(tout0.s0.tstart0.L0 + r0.ru.sack0.S11')

(6.53)

where

L0 = tout0.s0.tstart0

L1= tout1.s1.tstart1

S10' = rack1.su.s0.tstart0.(tout0.s0.tstart0.L0 + r0.ru.sack0.S11')

S11' = rack0.su.s1.tstart1.(tout1.s1.tstart1.L1 + r1.ru.sack1.S10')

(6.52) ==>

" 'rack1 precedes su precedes s0 precedes tstart0 precedes

('tout0 precedes s0 precedes tstart0 precedes

tout0 precedes s0 precedes tstart0 precedes

... ')'

or

'rack1 precedes su precedes s0 precedes tstart0 precedes

r0 precedes ru precedes sack0 precedes rack0 precedes su

precedes s1...' " (6.54)

where both constraint terms within the parenthesis will eventually occur in a fair E.

Ignoring intermediate actions in both terms of the constraint:

(6.54) ==>

" 'rack1 precedes s0 precedes s1 precedes rack1' " (6.56)

Applying the message number extraction function m to both parts of the constraint we get the message sequence:

mrack1,ms0,ms1',mr1'... (6.57)

In the sequence (6.57) mrack11 refers to the reception of an acknowledgement to a message with sequence number 1, preceding the sending of the message ms0, with sequence number 0. The message ms1' refers to the sending of the next message with sequence number 1, which follows the sending of the message with sequence number 0.

(6.56) and (6.57) ==>

" not ('s0 precedes rack1')" is always true.

Similarly, " not ('s1 precedes rack0')" is always true.

And therefore, property (6.51) is proved.

[end of proof]

2. Proof of Safety Invariant 6.2

" 'r0 precedes sack0'"

and

" 'r1 precedes sack1' "

Proof:

Again, ignoring intermediate actions from both terms of the partial constraint (6.51) we get:

" 's0' *or* 's0 precedes r0' " ==>

" 's0 precedes r0' " is always true.

Similarly, " 's1 precedes r1' " is always true. Therefore, proposition (6.2) is verified.

[end of proof]

3. Proof of Safety Invariant 6.3

" 's0 precedes r0 precedes s1' " is always true

Proof:

Ignoring intermediate terms not of interest the partial constraint (6.51):

==>

" 's0' *or* 's0 precedes r0 precedes s1' " is always true

==>

" 's0 precedes r0 precedes s1' " is always true

(6.58)

Applying the message extraction function m to the partial constraint (6.54) we get:

ms0(input),mr0(output),ms1(input),... (6.59)

the sequence (6.59) ==> minput = moutput + 1

Therefore, proposition 6.3 is proved.

[end of proof]

4. Proof of Safety Invariant 6.4

"The output sequence of messages is at all times an initial sub-sequence of the input sequence"

Proof:

Ignoring intermediate terms not of interest in the Partial Constraint (6.51) we obtain the constraint:

" 'us precedes ur' ",

applying the function m we get the sequence:

m(su(1)),m(ru(1)),m(su(2)),

==>

The input sequence is: m(su(1)),m(su(2)),

The output sequence is: m(ru(1))... where m(ru(1))=m(su(1))

Therefore, the output sequence is always an initial sub-sequence of the input sequence.

[end of proof]

Liveness

The liveness property:

Liveness Invariant 6.5

"Each message input into the system is eventually output"

is proved in the following.

The proof will be made in two parts:

(i) A message that is input is eventually output

(ii) A message that is output was previously input.

Proof of Liveness Invariant 6.5

The Partial Constraint (6.51) can re-written as:

' 'rack1 precedes su precedes s0 precedes tstart0 precedes

({ 'tout0 precedes s0 precedes tstart0 precedes tout0'}*

or

r0 precedes ru precedes sack0 precedes rack0 precedes su

precedes s1... precedes rack1')" (6.60)

The asterisked curly bracketed constraint in the expression is a notational shorthand to denote the fact that the system may loop through that sequence any number of times.

Since E is fair, either term within the parenthesis will eventually occur. That means if the curly bracketed term occurs now, then the other will eventually occur in the future and vice versa.

Therefore, (6.60) ==>

" 'rack1 precedes su precedes s0 precedes tstart0 precedes

({ 'tout0 precedes s0 precedes tstart0 precedes tout0'}*

precedes r0 precedes ru precedes sack0 precedes rack0 precedes su

precedes s1... precedes rack1')" (6.61)

This implies that that the curly bracketed term in (6.61) will occur 0 to n times before the action r0 occurs.

Therefore (6.61) ==>

" 'us precedes s0 precedes r0 precedes ur' " is eventually true (6.62)

(i) Therefore applying the function m on (6.62), for a given message 'm':

msu(input),ms0,mr0, mru(output),...

is eventually true (6.63)

Therefore, msu(input) = mru(output).

Therefore, a message that is input is eventually output.

(ii) Suppose, we are given a message 'm' output via the action 'ur'. Since, from (6.56) the strong constraint (6.62) is eventually true,

==>(6.63) (was already) is true for the same message,

==> mr0 (was already) is true for the same message,

==> ms0 (was already) is true for the same message,

therefore, the output message is the same as the one input by the action su.

A similar proof can be made for messages with sequence number 1

Note that it can be shown that there do not exist any branches within the two alternative paths which will eventually occur in system behaviour. This is due to the precedence constraints determining the sequence of events.

Therefore, the liveness Invariant 6.5 is proved.

[end of proof]

Chapter 7

CONCLUSIONS

This monograph is about the design and formal specification of distributed systems. The OSI Basic Reference model architecture is the source of most of the ideas and notions of protocols and distributed systems which have been analysed in the partial constraint design methodology developed in this treatise.

The motivation for this thesis emerged from complexity issues in the process algebraic design and specification of protocols and distributed systems. Protocol and distributed system behaviour is complex. Specifications of system behaviour in the currently available process algebras CCS, CSP,ACP and the standard FDT LOTOS prove to be extremely complex to comprehend[3],[4],[5]. This makes behaviour analysis of process algebraically described systems difficult. Attempts at validating the service provided by OSI protocols in particular and distributed system in general proves to result in combinatoric complexity. Using Bisimulation equivalence as a tool for verification has yet to overcome the hurdle of undecidability. This is due to the np-completeness of any procedure designed to show the equivalence of two algebraic processes with infinite behaviours , infinite recursions and infinite values of data.

Decidability therefore imposes the neccessity of developing design methodologies for distributed system specification which generate piece-wise complete Transition Systems such as FSTs. This then was the motivation for developing the partial constraint methodology.

The partial constraint methodology allows the designer to design and specify communication protocols and distributed systems and the services provided by them, in a simple piece-wise user-friendly manner. Use of partial constraint design methodology results in partial specifications which encapsulate only a subset of system behaviour requirements. The choice of requirements chosen for specification remain the designer's choice. Presumably, the designer will choose those requirements which are of primary interest in the tasks of validation and verification of specifications Partial specifications generate FSTs.

The first objectives of this thesis is to contribute by relating design objectives to design methodology for protocols and distributed systems behaviour. To characterise the essential features of distributed system behaviour, and provide a uniform framework of interpretation of existing behaviour models of distributed systems.

Chapter 1 characterises non-determinacy, concurrency and communications - essential features of distributed system behaviour. Chapter 2 identifies attributes which would make a behaviour model suitable for the formal specification of distributed systems. Existing behaviour models are interpreted uniformly as simulations of heterogenous algebras in chapter 3.

The second objective of the thesis is to contribute by formally defining the notion of *partial constraints* for use as design and specification tools.

In chapter 4, the new and original notion of partial constraints is introduced and formally defined. The relationship between events of a communication environment are formally defined in terms of the basic notions of distributed behaviour: non-determinism and communications. Illustrative examples show how simple partial constraints can

capture distributed behaviour - structure it and transform it into process algebraic specifications.

The third objective of this thesis is to contribute by developing a methodology for obtaining proces algebraic service specifications from protocol specifications.

Guidelines for the construction of constraints for protocol entities and the underlying service are given in Chapter 4. Mapping the partial constraints to partial process algebraic specifications, and then parallel composing three entities yields specification of the service

The fourth objective of this thesis is to contribute by showing how partial constraints can be used to validate specifications in the design-to-specification phase of the development trajectory.

The partial constraint design methodology is elaborated in chapter 5 with the formulation of process algebraic formulae for the specification, validation and verification of distributed system behaviour. The classic example of the Alternating Bit protocol is used as an example of a real protocol to illustrate the design methodology.

The fourth objective of this thesis is to contribute by showing how partial constraints can be used to verify specification in the design-to-specification phase of the development trajectory.

The essence of the partial constraint methodology is encapsulated by chapter 6. In this chapter the complextiy of validation and verification is identified and a validation procedure is elaborated. The validation procedure is applied to the connection establishment phase of the OSI transport protocol (class 0).

A verification procedure based on the classic Floyd-Hoare inductive assertion technique is also formulated in chapter 6. The Alternating Bit protocol is then verified for some safety and liveness properties by the application of this procedure.

The partial constraint method restricts the designer to use the basic structures permitted by partial constraints. However, the advantage is that the specifications can be developed partially and proved to be correct independant of future incremental extensions to the specifications.

Future work

Future extensions to this work will be an attempt to allow a more flexible structure of partial constraints, which is consistent with the basic partial constraint structure. It is hoped that structural extension of partial constraints will allow for proving more flexible, freer and hence more complex process algebraic specifications correct.

Another logical extension to the partial constraint method of design would be to develop the method for modularly extending partial constraint specifications such that complete specifications can be treated for verification analyses.

Another extension which is more logical and practical is the development of a test specification architecture for specifying and testing distributed systems, with tools to support distributed system development activities.

References

[1] R. Ahooja, J. Burmeister, J. de Meer and Axel Rennoch, " Open Systems Testing, a method, a language and a tool", proceedings of the second IWPTS conference held in Berlin, W. Germany, October 1989.

[2] A.V. Aho, J.E. Hopcroft, J.D. Ullman, "The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.

[3] R. Ahooja, "Towards Verification of Communication Protocol Specifications in LOTOS", Canadian Conference on Electrical and Computer Engineering, Vancouver, Canada, 1989.

[4] R.Ahooja, "Comparing Normal Forms in LOTOS and ESTELLE Specifications", 6th IFIP WS on Protocol Specification, Testing and Verification, Montreal, June, 1986, North-Holland, 1987.

[5] R. Ahooja and J. deMeer, "Protocol Validation Using Process Algebraic Guarded Recursive Equations", Proceedings of the COMNET '90, Budapest, Hungary, May 1990.

[6] K.A.Bartlett, R.A. Scantlebury, and P.T. Wilkinson, "A note on reliable full-duplex transmission over half-duplex links,"CACM,vol. 12, No. 5 1969.

[7] J.A.Bergstra, J.W.Klop, "Process Algebra: Specification and Verification in Bisimulation Semantics", in CWI Monographs 4, North Holland 1986, pp. 61-94.

[8] J.A. Bergstra, J.W. Klop, " Verification of an alternating bit protocol" by means of process algebra", Report CS-R8404, March 1984, Department of Computer Science, Centre for Mathematics and Computer Science, (CWI), NL.

[9] G.V. Bochmann, "A General Transition Model for Protocols and Communication Services", IEEE Transactionson on Communications, VOL. COM-28, No. 4, April 1980.

[10] T. Bolognesi, E. Brinksma, "Introduction to the ISO Specification Language LOTOS", Computer Networks and ISDN Systems, Vol/ 14, No. 1, 1987, pp. 25-59.

[11] G. Bochmann and C. Sunshine, "Formal methods in communication protocol design", IEEE Trans. Commun. vol. COM-28,pps. 624-631, Apr. 1980.

[12] T. Bolognesi, S A. Smolka, "Fundamental results for the Verification of Observational Equivalence: a Survey", Seventh IFIP international meeting on Protocol Specification, Testing and Verification, Zurich, May 5-8, 1987, (North-Holland 1988).

[13] M. Broy, "Specification and top down design and distributed systems", (invited talk), In H. Ehrig et al. (eds.) Formal Methods and Software Development, Lecture Notes in Computer Science 186, Springer 1985.

[14] E. Brinksma, R. Alderden et al., "Formal Notions in Conformance Testing", Proceedings of the 2nd International Workshop on Protocol Test Systems, Berlin(West), Germany, October 3-6, 1989.

[15] E. Brinksma, L. Logrippo, et. al.,"The OSI Transport Service and its Formal Description in LOTOS", COMNET 1985.

[16] R.M. Burstall, "Formal Description of Program Structure and Semantics in First-Order Logic", Machine Intelligence (Vol. 5),New York: American Elsevier, 1970, pp. 79-98.

[17] R.M. Burstall, "An Algebraic Description of Programs with Assertions, Verification and Simulation", Proceedings of CACM on "Proving Assertions about Programs", SIGPLAN Notices, 7(1), January 1972, pp. 7-14.

[18] P. Degano, U. Montanari "Distributed Systems, Partial Orderings of Events, and Event Structures", in Control Flow and Data Flow: Concepts of Distributed

Programming, M. Broy (ed) NATO ASI, Series Vol. F14, Springer-Verlag, Berlin, 1984.

[19] ISO/IEC JTC 1N, "Formal Description of ISO 8072 in LOTOS", 1989.

[20] E.W. Dyjkstra, "Notes on Structured Programming", in O.J. Dahl, E.W. Dykstra, C.A.R. Hoare, Structured Programming, Academic Press, New York, 1972, pp. 1-82.

[21] E.W. Dyjkstra, "Guarded Commands, Non-Determinacy and Formal Derivation of Programs", in CACM 18, 19775, pps. 453-457.

[22] H. Ehrig, B. Mahr, "Fundamentals of Algebraic Specification 1 - Equations and Initial Semantics", EATCS Monographs on Theoretical Computer Science, Vol. 6, W. Brauer, G. Rozenberg, A. Salomaa (eds.), Springer-Verlag, 1985.

[23] FDTs, "Formal Description Techniques", Proceedings of the First International Conference on Formal Description Tecniques", K.J. Turner (ed), Stirling, Scotland, 6-9 September, North-Holland, Amsterdam, 1989.

[24] ESTELLE, "The Formal Description Technique ESTELLE - Results of the ESPRIT/SEDOS Project", M. Diaz, J.P. Ansart, P. Azema, V. Chari (eds.), North-Holland, Amsterdam, 1989.

[25] LOTOS, "The Formal Description Technique LOTOS - Results of the ESPRIT/SEDOS Project", P.H.J. van Eijk, C.A. Vissers, M. Diaz (eds.), North-Holland, Amsterdam, 1989.

[26] CCITT, "Functional Specification and Description Language (SDL)", Recommendation Z100-104, VIII Plenary assembly, Malaga-Torremolinos, 1984

[27] R. W. Floyd, "Assigning meanings to programs", Proceedings of Symposia in Applied Mathematics XIX, pps 19-32, American Marthematical Society, 1967.

[28] J.B. Goodenough, S. L. Gerhart, "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975.

[29] J.A. Goguen, J.W. Thatcher et al., "Initial Algebra Semantics and Continuos Algebras", JACM 24, 1, pp. 68-95.

[30] J.V. Guttag, E. Horowitz et al., "Abstract Data Types and Software Validation", CACM, Vol. 21, No. 12, December 1978.

[31] B.T. Hailpern, "Verifying Concurrent Processes Using Temporal Logic", in Lecture Notes in Computer Science, No. 129, G. Goos and J. Hartmanis (eds), Springer-Verlag, Berlin, Germany 1982.

[32] P. B. Hansen, "Testing Multiprogramming Systems", Software Practice and Experience, April-June 1973,3, pp. 145-150.

[33] E.C.R. Hehner, L.E. Gupta, A. J. Malton, "Predicative Methodology", in Nato ASI Series, Vol. F36, Logic of programming and Calculi of Discrete Design, ed. M. Broy, Springer-verlag, Berlin, Heidelberg 1987.

[34] M. Herlihy, B. Liskov, "A Value Transmission Method for Abstract Data Types", ACM Transactions on Programming Languages and Systems, Vol. 4, No. 4, October 1982,pp. 527-551.

[35] C.A.R. Hoare, "Communicating Sequential Processes", Prentice-Hall International, Englewood Cliffs, New Jersey, U.S.A., 1985.

[36] C.A.R. Hoare, "An Axiomatic basis for Computer Programming", CACM 12,(10), Oct. 1969, pps. 576-580,583.

[37] W.E. Howden, "Algebraic Program Testing", Acta Informatica, Vol. 10,1978.

[38] J.E. Hopcroft, J.D. ULLman, "Introduction to Automata Theory, Languages, and Computation", Addison-Wesley, Reading, Mass., 1979.

[39] G.E. Hughes, M. J. Cresswell, "An introduction to Modal Logic", Methuen and Co. Ltd., London, 1968. Distributed in the United States by Harper and Row.

[40] ISO/IS7498, International standards Organisation, Open Systems Interconnections - Basic reference model 1988.

[41] ISO/IS8073, International standards Organisation, Open Systems Interconnections - Connection oriented transport protocol specification 1988.

[42] ISO/IS8072, International standards Organisation, Open Systems Interconnections - Transport Service definition 1988.

[43] ISO/IS8807, International standards Organisation, Open Systems Interconnections - "LOTOS - A Formal Description Technique based on the Tem[poral ordering of Observational behaviour" - 1988.

[44] B. Jonsson J. Parrow, "Deciding Bisimulation Equivalence for a class of Non-Finite-State Programs." In Proceedings of the Sixth Annual Symposium on Theoretical Aspects of Computer Science, 1989. Lecture Notes in Computer Science 349, pp. 421-433. Springer-Verlag, Berlin 1989

[45] J. de Meer, "Introduction t the Formal Description Technique LOTOS", in proceedings of the International Congress on Terminology and Knowledge Engineering, H. Czap, C. Galinski (ed)s), Indeks Verlag, Frankfurt, W. Germany.

[46] J. de Meer, K. P. Hasler,"OSI Transport Service considered as an Abstract Data Type", 1985, internal report, Hahn-Meitner Institute, Berlin GmBh.

[47] J. de Meer, "Derivation and Validation of Test Scenarios based on the Formal Specification Language LOTOS", 6th IFIP WS on Protocol Specification, Testing and Verification, Montreal, June, 1986, North-Holland, 1987.

[48] L. Lamport, ""Sometime" is Sometimes Not "Never": On the temporal logic of programs", Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, Jan 1980, pp. 174-185.

[49] L. Lamport, "Basic Concepts", in Distributed Systems, Lecture Notes in Computer Science, No. 190, M. Paul, H.J. Siegert (eds.), 1985, pp. 19-30

[50] L. Lamport, F.B. Schneider, "Formal Foundation for Specification and Verification", in Lecture Notes in Computer Science, No. 190, "Distributed Systems", G. Goos and J. Hartmanis (eds), Springer-Verlag, Berlin, Germany 1985.

[51] J.D. Lipson, "Elements of Algebra and Algebraic Computing", Addison-Wesley Publishing Company, Reading Massachussetts, 1981.

[52] Z. Manna, ""Properties of programs and the First-Order Predicate Calculus", JACM, 16(2), 1969, pp. 244-255.

[53] Z. Manna, "Mathematical Theory of Computation", New York: McGraw-Hill. 1974.

[54] Z. Manna, A. Pneuli, "Axiomatic approach to total correctness of programs", Acta Informatica, 3 pps 243-263, 1974.

[55] R. Milner, "A Calculus of Communicating Systems", Lecture Notes in Computer Science No. 92, G. Goos, J. Hartmanis (eds), 1980.

[56] R. Milner, "The Calculus CCS And Its Evaluation Rules", in Control Flow and Data Flow: Concepts of Distributed Programming, M. Broy (ed) NATO ASI, Series Vol. F14, Springer-Verlag, Berlin, 1984.

[57] R. Milner, "Fully abstract models of typed lambda-calculi", Thoretical Computer Science, 4, 1(Feb. 1977), pp 1-22

[58] R. Milner, "Calculi for Synchrony and Asynchrony" Th. Computer Science, 25, 1983, pp. 267-310.

[59] J.H. Morris, B. Wegbreit, "Program verification by subgoal induction", in Current Trends in Programming methodology, R.T. Yeh (ed), Vol. II, 1977, pp.197-227.

[60] E. Najm, "A verification oriented specification in LOTOS of the Transport Protocol", Seventh IFIP international meeting on Protocol Specification, Testing and Verification, Zurich,May5-8,1987 (North-Holland 1988).

[61] R. De Nicola, M.C.B. Hennesy, "Testing Equivalences for Processes", Th. Çomp. Sc. 34, 1984, pp. 83-133.

[62] S.S. Owicki, D. Gries, "Verifying properties of parallel programs: An axiomatic approach", Comm. of the ACM, 19 (5), pps 279-285, May 1976.

[63] D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", CACM, Vol. 15, No. 12, Dec. 1972, pp.1053-1058

[64] D. L. Parnas, "Designing Software for Ease of Extension and Contraction", in Proceedings of the third International Conference on Software Engineering, IEEE-ACM, May 1978, pps. 264-277.

[65] ISO/IEC JTC 1N, "Formal Description of ISO 8073 in LOTOS", 1990.

[66] C.A. Petri, "Concepts of Net Theory", Mathematical Foundations of Computer Science: Proceedings of Symposium and Summer School, High Tatras, Sept. 1973, Mathematical Institute of the Slovak Academy of Sciences, Bratislava, 1973, pp. 137-146.

[67] A. Pneuli, "The Temporal logic of programs", The 18th Annual Symposium on Foundations of Computer Science(providence, Rhode Island), pp. 46-57, IEEE, October, 1977.

[68] C.V. Ramamoorthy, S.F. Ho, "Testing Large Software with Automated Software Evaluation Systems", in Current Trends in Programming methodology, R.T. Yeh (ed), Vol. II, 1977, pp.112-150.

[69] L. Robinson, K. Levitt, "Proof Techniques For Hierarchically Structured Programs", in Current Trends in Programming methodology, R.T. Yeh (ed), Vol. II, 1977, pp.173-196.

[70] S. Schindler., U.Flasche, and D. Altenkruger, The OSA project: Formal specification of the ISO transport service, " Proceedings of the Computer Networking Symposium, National Bureau of Standards (USA), December 1980.

[71] G. Scollo, C.A. Vissers, A. Di Stefano, "LOTOS in Practice", Proc. IFIP 86, 10th World Congress, Dublin, Ireland, Sept. 1986,(North-Holland 1986) pp. 869-875.

[72] G. Scollo and V. Sinderen, "On the Architectural Design of the Formal Specification of the Session Standards in LOTOS", in G. Bochmann, B. Sarikaya (eds.) Protocol Specification, Testing, and Verification, V. North-Holland, 1986.

[73] N. Shiratori, H. Kaminaga et al., "A Verification Method for LOTOS Specifications and its Application", 9th IFIP WS on Protocol Specification, Testing and Verification, U of Twente, Enschede, NL, June 1989.

[74] N.V. Stenning, "A data transfer protocol", Comput. Networks, vol. 1, pp 99-110,Sept. 1976.

[75] C.A. Sunshine, "Formal modelling of communication protocols," in Computer Networks and Simulation, S. Shoemaker (ed.), North-Holland, 1982.

[76] C. Sunshine, "Four automated verification systems", 2nd IFIP WS on Protocol Specification, Testing and Verification, New York, North-Holland, 1982.

[77] C.A. Sunshine et al, " Specification and Verification of Communication protocols in AFFIRM using state transition models",in IEEE Transactions on Software Engineering, Vol. SE-8, No. 5, September 1982.

[78] J.W. Thatcher, E.G. Wagner et al., "Data Type Specification: Parameterization and the Power of Specification Techniques", ACM Transactions on Programming Languages and Systems, Vol. 4, No. 4, October 1982, pp. 711-732.

[79] A.J. Tocher, OSI Transport Service: A Constraint-Oriented Specification in Extended LOTOS, (ESPRIT/SEDOS/41), Nov. 85.

[80] C. A. Vissers and L. Logrippo, "The Importance of the Service Concept in the Design of Data Communications Protocols", in Protocol Specification, Testing, and Verification, V, M. Diaz (ed.) North-Holland, 1986.

[81] C.A. Vissers, G. Scollo, M. V. Sinderen, "Architecture and Specification Style in Formal Descriptions of Distributed Systems", Invited paper, Eighth IFIP

international meeting on Protocol Specification, Testing and Verification, Atlantic City, June 7-10,1988.

[82] C.A. Vissers, G. Scollo, M. V. Sinderen, "Architecture and Specification Style in Formal Descriptions of Distributed Systems", Revised version - unpublished.

[83] W.A. Wulf, R.L. London et al., "Abstraction and Verification in Alphard", In New Directions in Algorithmic Languages-1975, S.A. Schuman (ed), IRIA, 1976.

[84] R.T. Yeh, "Verification of programs by predicate transformations", in Current Trends in Programming methodology, R.T. Yeh (ed), Vol. II, 1977, pp.197-227.

[85] P. Zaforopulo et. al., "Protocol analysis and synthesis using a state transition model," in Computer Networks and Protocols, P.E. Green, Ed. New York: Plenum, 1983, pp. 645-670.

[86] J.R. Zhao, G. Bochmann, "Reduced Reachability Analysis of Communication Protocols", 6th IFIP IWS on Protocol specification, Testing and Verification, Montreal, June, 1986, North-Holland, 1987.

[87] H. Zimmermann, "OSI Reference Model, The ISO Model of Architecture for Open Systms Interconnections", IEEE Transactions on Comm. Vol. COM-28, n4, April, 1980, pp. 425-432.