

Applying Formal Techniques to the Design of Concurrent Systems

OCIEE-92-03

**Mark Vigder
Dept. of Systems and Computer Engineering
Carleton University
Ottawa Ont.
©1992**

**Applying Formal Techniques to
the Design of Concurrent Systems**

by
Mark Vigder, B.Sc, M.Eng

**A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Doctor of Philosophy**

Department of Systems and Computer Engineering

**Carleton University
Ottawa, Ontario
10 July 1992
©copyright
1992, Mark Vigder**

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis,

Applying Formal Techniques to the Design of Concurrent Systems

submitted by

Mark Vigder, B.Sc., M.Eng

Chair, Department of Systems and Computer Engineering

Thesis Supervisor

External Examiner

Carleton University

Abstract

This thesis applies formal techniques to the process of developing designs for systems constructed from a set of concurrent components. In particular, a technique for developing designs called *slicing* is identified and formalized. In the slice technique the designer identifies triggering events at the edges of a system, and then specifies the global behaviour patterns which ripple through the different components of the system as a result of these triggering events. The basic novelty lies in a combination of formal specification of global behaviour patterns with a formal representation of design structures that is not found in “pure” behavioural specification techniques and that is inspired by the way practical engineers design such systems. From this follow novel techniques for developing and analyzing complete designs for concurrent systems.

A method for representing slice behaviour of a system using the LOTOS specification language is developed. In order to facilitate design capture, a number of visual notations are developed to represent both structural and behavioural design. The visual notations allow the designer to express the design in a clear and intuitive manner while maintaining an underlying formal representation.

It is shown how slice based techniques can be used within a design process. Beginning with basic slices identifying the critical end to end behaviour of a system, specific techniques are developed for refining these slices to specify more complex behaviour. From these slice expressions, techniques are developed which allow a designer to analyze a design for correctness, and to assist in the specification and verification of individual component behaviours.

Examples and case studies are used to explore the applicability of the techniques and the visual notations. A discussion of the practical issues identifies how the slice style can be applied to large scale systems and discusses areas where automated tools can be used to assist the designer in the design process.

Acknowledgements

A thesis cannot be written in isolation, and there are many people whom I would like to thank for their assistance during the course of my research.

First, I would like to thank my advisor Dr. R.J.A. Buhr for his support, both professional and personal. Within his work can be found much of the basis for this thesis. Although I frequently resisted his suggestions (and usually I was wrong), his observations were almost always correct and this thesis has benefited greatly due to his advice.

Numerous faculty members from Carleton University and University of Ottawa have contributed to my research by providing advice, answering questions, and sitting on the various examination boards. For this I would like to extend a thank you to Gerald Karam, Moshe Krieger, Luigi Logrippo, Dorina Petriu, and Murray Woodside. Other people who have contributed to this thesis providing technical and moral support include Gregory Bond, Ron Casselman, Erik Dravnieks, Philippe Goldstein, and Serge Lamouret. I have without doubt left out some people from this list, for which I apologize.

Financial support for this research was provided by the Telecommunications Research Institute of Ontario.

I would also like to thank my children, Joseph and Samantha, who have never known their father as anything but a student. Without their never ending help I would likely have been able to finish this thesis a year ago.

Finally, a special thank you to Pamela Sweet. Without her encouragement to return to university after being away for many years, and without her patience and support, it is unlikely that I would ever have completed this work.

Contents

List of Figures	viii
Chapter 1	Introduction 1
1.1	Why Is Concurrent System Design Difficult? 4
1.2	An Effective Design Process for Concurrent Systems — A Possible Solution 6
1.3	Research Summary 9
1.3.1	Summary of Research Activities 9
1.3.2	Summary of Research Results 10
1.4	Organization of Thesis 12
Chapter 2	Conceptual Models and Formal Models of Concurrent Systems 15
2.1	Behavioural and Structural Design 15
2.1.1	Approaches to Structural Design 16
2.1.2	Approaches to Behavioural Design 16
2.2	Models of Behaviour 23
2.2.1	Behavioural Specification Languages 25
2.3	LOTOS: A Formal Modeling Technique for Behaviour 29
2.3.1	LOTOS Specification Styles 31
2.3.2	Verification in LOTOS 40
2.3.3	The Transition System Model 42
Chapter 3	The Design Problem: Behavioural Design by Slicing 50
3.1	Steps in the Design Process 51
3.1.1	Structural Design 53
3.1.2	Exploring Behavioural Design with Slices 53
3.1.3	Developing Component Specifications 59
3.1.4	Verifying Component Specifications Using Slice Expressions . 62
3.1.5	Structural Decomposition 62

3.2	Related Research	62
3.3	Chapter Summary	64
Chapter 4	Design Representation	66
4.1	Representing Structural Design	66
4.1.1	A Formal Representation of Structure	68
4.1.2	Discussion	80
4.2	Representing Behavioural Design	81
4.2.1	Behaviour of a Design Structure	82
4.2.2	Visual Notations	96
4.2.3	Structural Decomposition and Behaviour	106
4.2.4	Discussion	110
4.3	Chapter Summary	111
Chapter 5	Formalizing Slice Techniques	113
5.1	Relations Involving Slice Expressions	114
5.1.1	Other Possible Relations for Defining Slices	116
5.2	Developing Component Behaviour Specifications	119
5.2.1	The Syntactic Approach	120
5.2.2	The Semantic Approach	138
5.2.3	Discussion	142
5.3	Races and Critical Races	146
5.3.1	Constructing the Component Closure	148
5.3.2	Race Conditions	150
5.4	Verifying Component Specifications	157
5.5	Chapter Summary	161

Chapter 6	Application of the Slice Techniques: An Example	163
6.1	Specifying the Structural Design	164
6.2	Exploring Design Through Slices	167
6.3	Generating Component Specifications	186
6.3.1	Generating Component Specifications Syntactically	187
6.3.2	Generating Component Specifications Semantically	191
6.4	Verifying Component Specifications	193
6.5	Component Decomposition	195
Chapter 7	Practicality of Slice Techniques	199
7.1	Design Capture	199
7.2	Design Analysis and Verification	201
7.2.1	Verifying the ext Relation	202
7.2.2	Extracting Component Behaviour from Slices	204
7.3	Design Execution	208
7.4	Implementation	212
7.5	Chapter Summary	212
Chapter 8	Conclusions	214
8.1	Contributions	214
8.1.1	Formalization of a Slice Technique	214
8.1.2	Formalizing the Representation of Structure	216
8.1.3	Formally Linking Structure and Behaviour	216
8.1.4	Visual Methods for Design Capture	217
8.2	Conclusions	218
8.2.1	Applicability of Slices to an Effective Design Process	218
8.2.2	LOTOS as a Behavioural Design Language	219
8.2.3	Visualizing Structural and Behavioural Design	220
8.3	Future Research	220
8.3.1	Tool Support	221
8.3.2	Using Slices	221
8.3.3	Extensions to Structural Representation	222

Appendix A	Summary of Notation and Conventions . . .	223
Appendix B	Summary of LOTOS Operators	226
Appendix C	Proof of Theorems	227
Appendix D	Case Study – A Telephony System	237
D.1	Problem Description	237
D.2	Structural Specification	238
D.3	Slice Specification	243
D.3.1	A Simple Slice — Successful Call Completion	243
D.3.2	Extending Slices	248
D.3.3	Combining slices	257
D.4	Component Specifications	273
Appendix E	Definition of phi_C Transform	278
Bibliography	284

List of Figures

Figure 2.1	The structural design process.	17
Figure 2.2	The design of a system.	18
Figure 2.3	Describing behaviour of interconnected components.	20
Figure 2.4	The design process: hybrid behaviour description.	23
Figure 2.5	Specification styles: extensional and intensional.	26
Figure 2.6	Specification styles: operational and constraint oriented.	27
Figure 2.7	Structure of Producer/Consumer system.	33
Figure 2.8	Example of constraint style in LOTOS.	34
Figure 2.9	Example of state oriented style in LOTOS.	36
Figure 2.10	Example of state oriented style in LOTOS – visual representation.	37
Figure 2.11	Example of resource oriented style in LOTOS.	39
Figure 2.12	Example transition system B_1	43
Figure 2.13	Example transition system.	46
Figure 2.14	Transition system B_2 is an extension of B_1	49
Figure 3.1	Superimposing a slice expression on a structure.	51
Figure 3.2	Extending simple slice expressions into more complex slice expressions.	55
Figure 3.3	Defining slice segments.	56
Figure 3.4	Constructing the slice $S1'$ from segments $S1_a$, $S1_b$ and $S1_c$	57
Figure 3.5	Combining slice expressions to create a new slice expression.	58
Figure 3.6	Process for combining slices.	60
Figure 3.7	Extracting component behaviour from a slice expression.	61
Figure 4.1	Producers/consumers problem – structural design specification.	75
Figure 4.2	Graphical icons for structural design.	76
Figure 4.3	Example of a hierarchical decomposition of components.	78
Figure 4.4	Producers/consumers problem – Structural decomposition.	79
Figure 4.5	Producers/consumers problem – Structural decomposition tree.	79
Figure 4.6	Design structure example — a buffer.	83
Figure 4.7	Observations at IPs showing possible traces.	83
Figure 4.8	Observations at IPs showing possible refusals after a trace.	85

Figure 4.9	Example interactions for the producer/consumer structure. . .	87
Figure 4.10	Example of resource oriented style in LOTOS.	89
Figure 4.11	Structure of system whose behaviour cannot be directly represented in a resource oriented style.	90
Figure 4.12	LOTOS resource oriented specification for structure of figure 4.11	93
Figure 4.13	Example of a slice style for a the producer/consumer example.	94
Figure 4.14	Producer/consumer slice example in a visual notation.	95
Figure 4.15	Timeline example for producers/consumers structure including parameters and exit values.	99
Figure 4.16	Graphical icons for timeline operators.	101
Figure 4.17	An example of combining timelines.	102
Figure 4.18	Example of a state transition diagram.	103
Figure 4.19	Example of a state transition diagram with parameters and exit values.	105
Figure 4.20	Resource oriented specification example.	108
Figure 4.21	Resource oriented specification example with decomposed components.	108
Figure 5.1	Example structure to illustrate problem with the bisimulation relation.	118
Figure 5.2	Transition systems which are failure equivalent, but which are not bisimilar.	118
Figure 5.3	Comparison of refusals and traces as a basis for relating slice expressions and system behaviour expressions.	118
Figure 5.4	Extracting component expressions from slice expressions. . .	120
Figure 5.5	Syntactic approach to extracting component behaviour from slice expression.	121
Figure 5.6	Semantic approach to extracting component behaviour from slice expression.	121
Figure 5.7	Slice expression for the consumer/producer structure.	123
Figure 5.8	Behaviour of component cons1 extracted from slice expression by phi.	126
Figure 5.9	Simplified behaviour of component cons1	127
Figure 5.10	Component behaviour of cons1 with external component 'anonymous'.	127

Figure 5.11	Term rewriting used to simplify component expressions. . . .	128
Figure 5.12	Slice expression using choice operator.	137
Figure 5.13	Projecting a timeline trace onto interactions of component c. .	139
Figure 5.14	Generating the failures(S_C) from failures(S)	141
Figure 5.15	Nondeterministic slice expression.	145
Figure 5.16	Illustration of race condition.	147
Figure 5.17	Constructing the component closure.	149
Figure 5.18	Structure exhibiting race condition.	151
Figure 5.19	Critical race example.	153
Figure 5.20	Slice expression showing two customers buying gas.	154
Figure 5.21	Component closure for gas station example.	155
Figure 5.22	Trace exhibiting a critical race.	156
Figure 5.23	Composing component specifications for a subset of the components.	160
Figure 6.1	Structural design of a simple telephone system.	165
Figure 6.2	Slice showing simple connection establishment: telephony example.	166
Figure 6.3	Timelines defining different phases of a connection establishment: telephony example.	169
Figure 6.4	Refining the simple call connect by extension.	170
Figure 6.5	LOTOS specification of extendConnect.	172
Figure 6.6	State machine defining the constraint that a telephone cannot be engaged in more than one call simultaneously.	174
Figure 6.7	LOTOS slice expression showing user(1) calling user(2) and user(3).	177
Figure 6.8	Combining slices.	180
Figure 6.9	Combining slice expressions by removing shared interactions: telephony example.	181
Figure 6.10	State machine used to synchronize slices: telephony example.	182
Figure 6.11	LOTOS expression for twoCalls: telephony example.	185
Figure 6.12	Component expression for switch syntactically derived from twoCalls.	190
Figure 6.13	List of traces and offers for component switch derived from slice expression twoCalls.	192
Figure 6.14	Decomposed switch component.	196

Figure 7.1	Design structure used to determine performance of ISLA interpreting a resource oriented specification.	210
Figure 7.2	Performance of ISLA interpreting a resource oriented specification.	210
Figure 7.3	LOTOS specification used to determine performance of ISLA interpreting a resource oriented specification.	211
Figure A.1	Set notation.	223
Figure A.2	Logical quantifiers.	223
Figure A.3	Sequences.	224
Figure A.4	Lists.	224
Figure A.5	Definitions of properties of transition system B.	225
Figure A.6	Interaction sets.	225
Figure D.1	Structure diagram for telephony example — phoneset and central switch.	240
Figure D.2	Structure diagram for telephony example — central switch subcomponents.	241
Figure D.3	Structure diagram for telephony example — simplug subcomponents.	242
Figure D.4	Simple slice: system level components.	245
Figure D.5	Simple slice: switch level components.	246
Figure D.6	Simple slice: simplug level components.	247
Figure D.7	Segmenting a slice: switch level components.	253
Figure D.8	Segmenting a slice: switch level components.	254
Figure D.9	Segmenting a slice: switch level components.	255
Figure D.10	Constructing a slice representing concurrency: branchSwitch.	256
Figure D.11	Removing shared interactions switch level: defining process busy_p by removing interaction send.	259
Figure D.12	Removing shared interactions switch level: a process defining connection establishment.	260
Figure D.13	Synchronizing process for switch level: single call per phoneset.	261
Figure D.14	Synchronizing process for switch level: busy signal returned if destination phone in use.	262
Figure D.15	Synchronizing process for switch level: single call per internal line.	263

Figure D.16	Synchronizing process for switch level: busy signal returned if no internalLine component available.	264
Figure D.17	Combining slices switch level: behaviour concurrentConnect	272
Figure D.18	Expression phi_alloc derived from slice expression concurrentConnect.	277

Glossary

Component. An element of the structural design which represents an artifact which can be implemented independently of the other parts of the system.

Component behaviour expression. A behaviour expression which specifies the behaviour of a specific component.

Composing component specifications. A method of joining a set of *component behaviour expressions* into a single *resource oriented specification* which is an expression representing the overall system behaviour.

Concurrent system. A system in which the various parts of the system execute independently and concurrently.

Constraint style specification. A style of specifying behaviour which identifies individual constraints of a system. The overall specification is then the conjunction of the individual constraints.

Constructing a slice from segments. Using behaviour expression operators to join expressions representing *slice segments* into a behaviour expression representing a new slice expression.

Extensional specification. A behaviour specification which describes a system's behaviour entirely in terms of interactions between the system and its environment.

Intensional specification. A behaviour specification which describes a system's behaviour using interactions internal to the system, as well as interactions between the system and its environment.

Interaction point. An element of the structural design which represents a place where components can synchronize and communicate with each other.

Monolithic style specification. A style of specifying behaviour which identifies explicit event sequences which must occur.

Primitive component. A component which is not structurally decomposed into subcomponents.

Refining slices. Developing new slices from a set of existing slices. Refinement is done either by *slice combination* or *slice extension*.

Resource oriented specification. A style for representing the behaviour of a concurrent system, where the behaviour of each component (i.e., resource) is represented as an expression, and the individual component behaviour expressions are then *composed* into an expression representing the overall system behaviour.

Segmenting slices. Dividing a slice expression into smaller subexpressions.

Slice combination. A method of refining slice expressions by combining them in parallel to represent concurrent behaviour. Combining slices usually requires a *synchronizing process* to identify how the concurrent slices interact with each other.

Slice extension. A method of refining a slice by adding new behaviour to it. Slice extension is done by means of *segmenting the slice* followed by *constructing a new slice from segments*.

Slice style specification. A style for representing the behaviour of a concurrent system, where the subexpressions within the slice style specification represent behaviour patterns involving many components of the system.

State oriented specification. A style of representing behaviour by means of a state transition system.

Synchronizing process. A behaviour expression which constrains the behaviour of two slices when they are combined

Chapter 1: Introduction

This research is concerned with developing *techniques* which can be used to support an *effective process* for generating the *design of concurrent systems*. In order to understand this statement it is necessary to ask four questions:

1. What is a *concurrent system*?
2. What is a *design* of a concurrent system?
3. What is an *effective process* for concurrent system design?
4. What are *techniques* which support an effective process for concurrent system design?

What is a concurrent system?

A concurrent system is a system which is constructed from a number of separate components where the components execute concurrently.

In the modern age, computers have progressed from relatively simple data processing tools to large systems interacting in complex ways with their environments. Rather than simply receiving data, processing it, and returning a result, many computer based systems now must monitor in an ongoing way different stimuli from the environment and determine appropriate actions to take depending on these stimuli. Such systems are often referred to as *reactive systems* since rather than simply processing data they must interact with the environment in an ongoing and continuous manner.

Reactive systems are often considered as *event driven* rather than *data driven*. An event driven system is one where the system receives events from the environment, determines appropriate responses, and issues these responses to the environment in a timely manner. Although data processing is involved, the difficulty and complexity lies in determining correct event sequences and appropriate responses to external events.

Reactive systems are generally constructed from a number of smaller and simpler *components*. Each component is a self contained unit and can be implemented as a separate artifact independent of the other components of the system. The system is built by combining these individual components into the overall system.

Often the components execute concurrently and independently of each other. For example in a distributed system, where rather than residing on a single computer the components are distributed over a number of physically separate computer systems, the different computer systems are executing concurrently with each other. Thus the components are executing, communicating and synchronizing where necessary.

What is a concurrent-system design?

A concurrent system design is a description of a system which specifies the structure of the system (i.e., the components and their interconnections) and the behaviour of the system (i.e., how the system interacts with its environment and how the concurrent components interact with each other). The design must be at a sufficiently detailed level to allow an implementor to build the system according to the design specification.

A *design* of a system is a description which is sufficiently detailed to allow an implementor to build the system. For concurrent systems design, what constitutes a ‘sufficiently detailed description’?

The design of concurrent systems can be viewed as proceeding along two axes: *structural design* and *behavioural design* [10,9,36].

Structural design specifies the components from which a system is constructed and how these components are connected to each other.

Behavioural design specifies how the system behaves, both in terms of how the system interacts with its environment as well as how the components of

the system interact with each other. Behaviour can be further subdivided into *functional behaviour* and *temporal behaviour*:

- The *functional behaviour* of a system is characterized by the data processing performed by the system.
- The *temporal behaviour* of a system is characterized by how the system behaves through time, for example when data functions are computed, when concurrent components synchronize with each other, etc.

A behavioural design of a concurrent system therefore must describe the functional and temporal behaviour of a system both in terms of how the system interacts with its environment and in terms of how the concurrent components interact with each other.

What is an effective process for concurrent-system design?

A process for concurrent system design is a set of steps which a designer uses to generate a design from a description of the requirements of a system. The process is effective if at the end of the process the designer has produced a correct, understandable, and robust design.

Once the general requirements of a system have been identified, how does a designer proceed to generate the design of the concurrent system? The requirements identify the constraints which the system must satisfy, e.g., ways in which the system must control its environment, a distribution topology imposed by the underlying network, etc. The task of the designer is to construct a design of the system which satisfies all the constraints imposed by these requirements. How does the designer approach this problem? Clearly, once the design process is complete the designer must have identified all the components from which the system is constructed, as well as having specified all possible valid behaviours of the system. A design process is the set of steps a designer uses to arrive at the design; the process is effective if the resulting design is clear, understandable, and satisfies all the original behavioural requirements.

What are techniques which support an effective process for concurrent system design?

Techniques which support an effective process for concurrent system design are specific methods which are used to implement the design steps.

An effective design process involves a set of design steps. Each design step involves the synthesis of new design information, capture of this information, and analysis of the information for correctness. The specific methods the designer uses to synthesize, capture, and analyze design information are the design techniques. For example, the design process may require the designer to specify the behaviour of a specific component of the system. The designer may then require techniques to determine the behaviour of the component, a technique to capture the behavioural design (i.e., a behavioural specification language), and techniques to analyze the behavioural specification for correctness.

Once a design process has been decided upon, specific techniques can be selected which support the design process. Where appropriate these techniques can be based upon a suitable mathematical formalism; other techniques will remain informal. Where possible, techniques should be supported by a set of automated tools which assist the designer in applying the techniques to the design steps.

1.1 Why Is Concurrent System Design Difficult?

Generating correct designs for concurrent systems has proven to be one of the most difficult problems in the field of computer system engineering. The reason for this stems from the inherent complexity of concurrent systems. This complexity arises due to a number of factors.

Multiple threads of control.

In sequential programs, the complexity of a system arises primarily due to the data functions computed and the algorithms used to compute those functions.

Because of the deterministic nature of sequential systems, the system can be understood and analyzed by stepping through the algorithm.

However in concurrent systems where there are a number of different threads of control, the system cannot be understood and analyzed by stepping through the system behaviour. Different activities are occurring concurrently within different parts of the system. The sequence of events in concurrent systems is nondeterministic; race conditions occur and synchronization is required between the different concurrent threads of control.

In addition concurrency can introduce into a system potential problems not found in sequential systems, e.g., deadlock, starvation and critical races [51,53]. Many of these problems are very subtle involving interactions between many different control threads and thus result in erroneous designs being generated.

Large numbers of interacting components.

Concurrent systems are typically constructed from a large number of components. In order that the system satisfies the requirements it is necessary that all these components operate and interact with each other correctly. To understand the design of the system it is not sufficient to understand the behaviour of each component in isolation. Rather it is necessary to know how the components interact with each other to provide the system functionality.

Unfortunately, given the complexity of many concurrent systems, understanding the behaviour of each individual component is difficult. Trying to understand the overall system behaviour by looking at the behaviour of each individual component and how these components interact, or are supposed to interact, is an even more formidable problem. In many cases, a better understanding of the problem can be gained by viewing the problem at the system level, where behaviour is specified across component boundaries, rather than at the component level where the behaviour of each component is specified independently of the other compo-

nents. However a system level view requires some means of factoring the system behaviour in a way which can be understood by designers.

1.2 An Effective Design Process for Concurrent Systems — A Possible Solution

Given the complex and difficult problem of designing concurrent systems, what can be done to assist the designer in generating correct designs? This research has investigated a number of techniques which are intended to assist the designer in generating designs. The main hypotheses investigated are the following.

Expressing composite behaviour of interacting components should be done initially at the system level rather than beginning by defining the behaviour of individual components.

When specifying the behaviour of interacting concurrent components, one approach is to describe the behaviour of each component in isolation. The system behaviour is then given as the composition of the individual component specifications. Although this is the traditional technique used in many design methods it is often difficult to apply in practice due to the complex and subtle interactions required between components. A different technique which can be applied is to allow the designer, at the design level, to describe behaviour patterns involving sets of interacting components. Thus rather than looking at each component in isolation, the designer begins the behavioural design step by identifying a behaviour pattern of the system requiring cooperation between a number of components. The designer then expresses this system wide behaviour pattern involving a set of components before identifying the behaviour required of each individual component. The advantages to be gained by this approach include the following:

- Much of the complexity of concurrent systems arises due to the interactions of many concurrent components which are cooperating to achieve an overall system requirement. By describing behaviour of the system in terms of how these components interact along critical threads spanning the system, the designer is taking a more global and intuitively cleaner approach to behaviour specification and has a higher probability of generating a correct design.
- Once a designer has a clear understanding of how the components must interact with each other to achieve the system requirements, the designer can use this information to assist in the correct specification of individual components which when composed will be expected to correctly satisfy these requirements.

The design process should allow for incremental extension of both structural and behavioural design.

Designers typically work by sketching out major characteristics of a design and then filling in details. Any design process should support such a technique, particularly when specifying complex behaviour patterns. For example when specifying the behaviour of a set of concurrent components a designer will typically begin with descriptions of the major behaviour patterns of the system, ignoring issues such as error conditions, concurrency, etc. These initial behaviour specifications can be used to verify that the design is capable of supporting the major functionality required of the system. Once the major behaviour patterns are identified, the designer can then proceed to refine these patterns, identifying variations, showing the concurrency, etc. Each refinement provides more detail to the behaviour.

Formal techniques should be used wherever possible.

One thesis of this research is that formal methods can and should be used as part of the engineering discipline applied to concurrent system design.

Complex concurrent systems require a design specification which is unambiguous and correct. Formal techniques can assist in generating such a specification. In order to increase the reliability of large scale concurrent systems it is necessary to have strong methods for *synthesizing* systems design, as well as for *analyzing* the design for correctness. If synthesis and analysis are to be done in a rigorous and reliable manner then it is necessary to have a precise and unambiguous representation of the design; formal techniques can provide such a precise and unambiguous representation. Although formal methods are not widely used in industrial applications their use could assist in generating more robust designs [33]. Without formalization, notations used for design capture will remain ambiguous and open to different interpretations by different observers. As well, methods used for synthesis and analysis cannot be proved correct and therefore are suspect.

Visualizations of complex designs should be used wherever possible.

Concurrent system design requires a large amount of complex information to be understood by the designer; proper use of visualization assists a designer in organizing, understanding and analyzing this information. At each step of the design process, specific design information generated by the designer must be captured. This could for example be structural information identifying components and their interconnections, or behavioural design showing how sets of components interact to provide some system functionality. Capturing this information can be greatly assisted by visualization; for example a designer can often “see” behaviour patterns moving through components. Rather than forcing a designer to transform these visual representations into a textual language, it is often more natural for a designer to draw the patterns directly on the design structure; translation to the formal, textual language can then be done as a second step.

1.3 Research Summary

The main objective of this research is to develop and formalize techniques which can be applied to the process of developing designs of concurrent systems. The techniques developed are based on the concept of specifying the behaviour of a set of concurrent components using *slices* where a slice is defined as follows.

Definition: Slice

A *slice* is a path traced through a set of components which shows the temporal sequencing of stimuli and responses in which the components must engage in order to satisfy the overall, end-to end system requirements.

The techniques developed by this research can be applied to the following steps of a design process (the design process considered by this thesis is described more fully in §3.1):

- *Identifying structural design.* As part of the design process a designer must identify the components and their interconnections.
- *Specifying system behaviour by 'slices'.* A slice, in its simplest form, identifies a temporal sequence of actions in which the components of the system engage. Beginning with the simple slices showing the major behaviour patterns, a designer refines these simple slices into more complex behaviour patterns.
- *Developing component specifications.* In order that components can be implemented independently of each other it is necessary to have independent behavioural specifications of each component. Thus as part of the design process a designer must create the component behaviour specifications.

1.3.1 Summary of Research Activities

The main activities which were undertaken as part of the research are the following:

- *Formalization of a ‘slice style’ of specification.* The concept of a ‘slice style’ of specification was formalized. Specific techniques for refining a slice style of behaviour specification were defined. Techniques for using slices to analyze and verify designs were explored.
- *Development of a representation of design structure.* In order to investigate the slice style of specification, a method of formally representing structural design was developed.
- *Use of LOTOS as a behavioural design language.* LOTOS was integrated with the structural design representation and used as a basis for representing behavioural design.
- *Case studies performed.* A number of examples and case studies were developed. Structural and behavioural designs for these case studies were worked out. The behavioural designs were validated by executing them on a LOTOS interpreter.
- *Development of visual notations.* In order to facilitate design capture, notations for visual design representation were developed which allow a designer to work with design level abstractions which are based on formal techniques.

1.3.2 Summary of Research Results

The original contributions which have been achieved during this research are the following.

Design representation.

To represent design structure, a formalism is developed which defines design structure in terms of the components and component interconnections (§4.1). The structural design representation allows for the hierarchical decomposition of components. The novelty of the approach lies primarily in the framework this provides for using LOTOS as a means for specifying behaviour of a concurrent system using slice expressions.

To represent the behavioural design, the formal language LOTOS is integrated with the structural design (§4.2). In using LOTOS as the behavioural design languages, a number of issues are addressed (§4.2.1.2) including:

- How to integrate LOTOS with the structural design.
- How to represent the behaviour of arbitrary networks of components where each components behaviour is represented as a LOTOS behaviour expression.
- How to use LOTOS to express both the slice behaviour of sets of components as well as the behaviour of an individual component.
- How to use LOTOS expressions at different levels of the structural design hierarchy (§4.2.3).

Refinement and analysis techniques using slices

Methods for representing slice expressions using the LOTOS behavioural specification language integrated with a structural design representation are developed (Chapter 4).

Techniques for refining slices by *slice extension* or *slice combination* are identified (§3.1.2). Slice extension refines a slice expression by identifying variations of the slice which result in different possible outcomes, and then adding these extensions to the original slice expression. Slice combination refines behaviour by combining two or more slice expressions to show their concurrent behaviour. Examples of slice refinement are provided in Chapter 6, Appendix D and [26]. The refinement techniques identified are independent of the language used to represent behaviour. The examples and case studies of the thesis show how the refinement techniques can be applied using LOTOS as the behavioural design language.

Given a slice expression, techniques for determining the behaviour of individual components from the slice expression are developed. Extracting component behaviour from a slice expression can be done *syntactically* by transforming the LOTOS slice expression into an expression representing the behaviour of an in-

dividual component (§5.2.1, Appendix E); or *semantically* by executing the slice expression and from the execution model determining the behaviour of the component (§5.2.2). Once the component behaviour has been extracted from the slice expression, this can be used as a means of developing the component specifications (§5.2), verifying that the component specification satisfies the behaviour implied by the slice expression (§5.4), or analyzing the design for potential critical races (§5.3).

Visual notations.

To assist the designer in capturing and understanding behavioural and structural design, a number of visual notations are developed. A notation for structural design defined in §4.1.1.1 allows a designer to enter the structural design almost entirely in the visual notation.

Visual notations for behaviour are developed which are applicable at different stages of the design process (§4.2.2). A notation called *timelines* is defined which permits a designer to define simple, sequential slice expressions, and assists a designer in behaviour refinement by slice extension. A visual notation for state transition diagrams is defined which is used to define synchronization constraints when refining behaviour by slice combination. Both of these visual notations can be automatically translated into LOTOS process definitions. (In the examples included in this thesis, only the state transition diagrams were automatically translated into LOTOS by means of a tool developed as part of this research; other translations were all performed manually.)

The original contribution of this thesis regarding the visual notations is the precision provided to the visual notation by the underlying formal model. This allows a designer to precisely specify designs using the visual notation, and allows designs which have been developed in the visual notation to be analyzed using formal techniques.

1.4 Organization of Thesis

Chapter 2 surveys a number of issues in concurrent system design. Possible approaches to the problem of specifying the behaviour of concurrent components are discussed. As well methods for behavioural design specifications are presented, and LOTOS [46] is evaluated as a behavioural design language.

Chapter 3 of this thesis puts forward a method for developing concurrent systems based on the concepts of hierarchical decomposition of structure, slice expressions to describe behaviour involving many concurrent components, and behaviour specification by incremental refinements, whereby a designer begins by describing the major behaviour patterns which must be observed, and then refining these initial behaviours to describe more complex patterns.

In order to use the proposed design method, it is necessary to have a formal means of representing structural and behavioural design. Chapter 4 develops a representation for structural design and integrates LOTOS with the structural design representation in order to represent design behaviour. A number of visual notations are developed which can be used during the design process.

Chapter 5 provides a formal foundation for the slice style of behaviour specification. Formal techniques for generating component specifications from slice expressions, analyzing designs, and verifying component specifications are developed.

Chapter 6 illustrates, by example, the integration of the techniques developed in this research into a process for concurrent system design. A structural design is specified, including structural decomposition of components. Behaviour is specified through a combination of slice expressions and behaviour specifications for individual components.

Chapter 7 discusses practical issues in applying the design techniques to large, industrial scale problems. Manual application of the techniques, possible tools to

support the techniques, and the feasibility of automatic verification and analysis are looked at.

Finally chapter 8 summarizes the work performed, discussing results and conclusions, and identifying future areas of research.

Chapter 2: Conceptual Models and Formal Models of Concurrent Systems

This chapter discusses a number of issues and techniques related to modeling concurrent systems.

Section 2.1 looks at modeling behavioural design and modeling structural design, the relationship between the two, and possible approaches which can be used in arriving at a complete system design.

Section 2.2 discusses a number of issues regarding the representation of the behaviour of a system. A number of properties of specification languages and models are identified and discussed.

Finally, section 2.3 looks at the specification language LOTOS and its applicability to the problem of modeling behavioural design.

2.1 Behavioural and Structural Design

Concurrent systems are constructed from a number of different components, with multiple threads of control and complex interaction sequences required between the components. In order to make the design problem tractable it is necessary to approach the problem through a separation of concerns, dividing the design problem into orthogonal concerns and then looking at each one in turn. Although there are many concerns of interest during the design of concurrent systems, this research approaches the design by dividing the design process along *structural* and *behavioural* lines. A design process for concurrent systems could then involve alternating between structural design and behavioural design. How the designer alternates between these two design concerns and the order in which the design is performed depends on the nature of the problem and the preference of the designer.

2.1.1 Approaches to Structural Design

The structural design process considered in this thesis involves the identification of components, determining how the components are interconnected, and the decomposition of components into subcomponents. The process begins with the designer identifying the boundaries of the system. This involves determining what functionality is included within the system and where the system interacts with its environment, i.e., the *interaction points*. This is illustrated by the leftmost diagram of figure 2.1, where the rectangle represents the system under design and the lines extending from the rectangle represent the means by which the system and its environment interact.

Once the system and its interaction points have been defined the designer identifies the components from which a system is constructed and the interconnection of these components (centre diagram of figure 2.1). Each of these components can encapsulate further structure and thus may be further decomposed (rightmost diagram of figure 2.1).

A structural design is complete once the designer has hierarchically decomposed the system into components which cannot be further decomposed. These are the primitive components of the system.

Definition: Primitive component.

A *primitive component* is a component which can be implemented directly without further decomposition into a network of internal components.

2.1.2 Approaches to Behavioural Design

The behavioural design process considered in this thesis involves specifying how a system and its environment interact with each other. This can be either the *functional behaviour* of a system which identifies the data values input to the system and the data values output by the system; or the *temporal behaviour* of

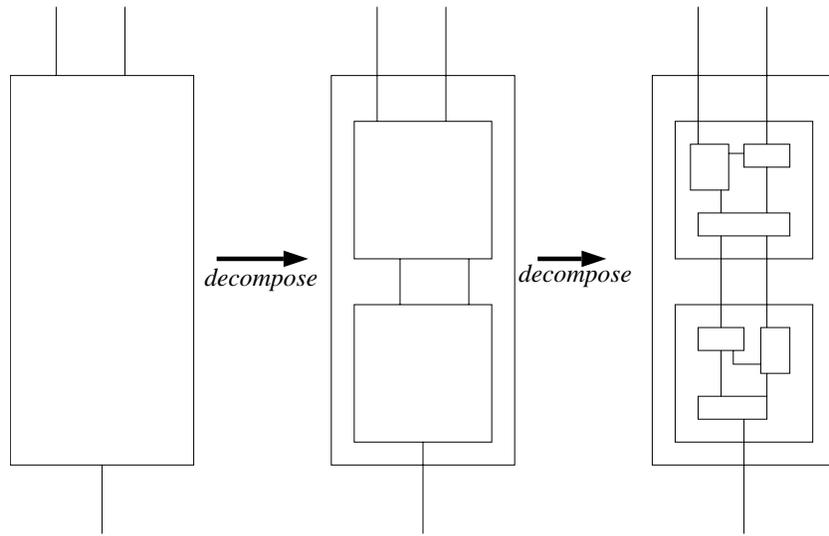


Figure 2.1 The structural design process.

the system which identifies temporal orderings on the interactions between the system and its environment. Behavioural design can be performed at any level of the structural design hierarchy, for example:

- The behaviour of the system can be specified ignoring any internal components, i.e., the ‘black-box’ behaviour of the system.
- The behaviour of a set of non-primitive components can be specified.
- The behaviour of the primitive components of the system can be specified.

The behavioural design of a system is complete once the designer has identified the primitive components of the system and completely defined the behaviour of these components. Graphically the design process is illustrated in figure 2.2. The icons of the figure represent the following:

- Rectangles without inscribed circles represent components. The dotted rectangles represent components external to the system being designed.

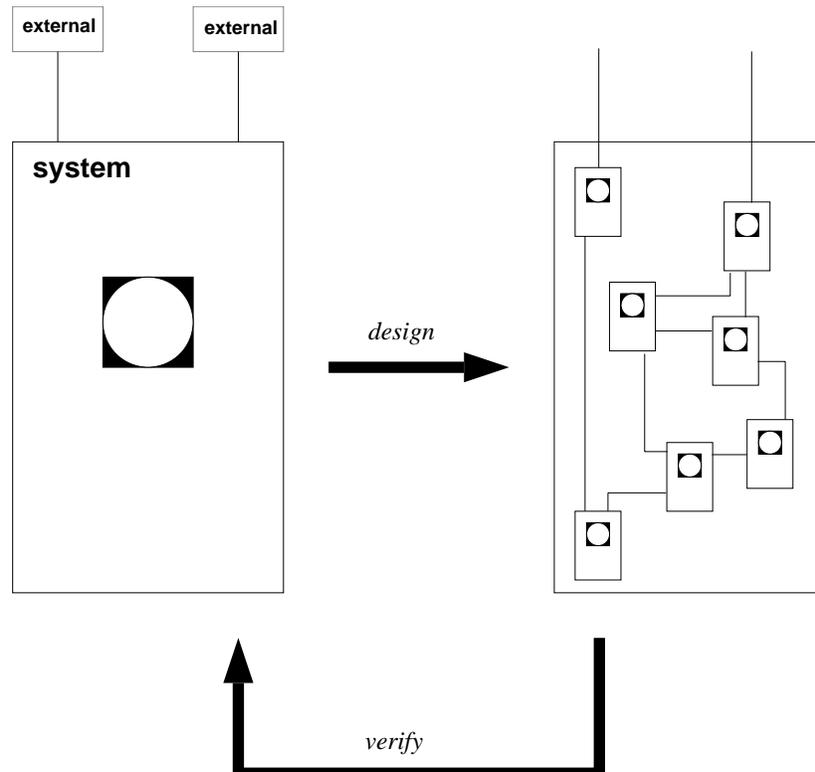


Figure 2.2 The design of a system.

- Circles inscribed in a square represent behavioural descriptions. It is not assumed that these behavioural descriptions need be complete or formal. Nor is it assumed that behavioural descriptions represent components, i.e., implementation artifacts.
- Lines joining component rectangles represent the means by which the components can communicate and synchronize with each other. At this stage, no assumption is made as to what these lines represent; they could be simple synchronization mechanisms or they could represent complex protocols between the components.

Examples of design processes which begin by identifying system behaviour, decomposing a system into components, and then factoring the system behaviour between the components include Hatley/Pirbai[37], Statemate[35], and Ward/Mellor[86]. These methods specify system behaviour by a combination of data flow (i.e., functional behaviour) and control flow (i.e., temporal behaviour). Once the system is structurally decomposed, the behaviour is then allocated to the different components.

As is illustrated in figure 2.2, the arrow labeled '*design*' requires a designer to identify components from which a system is constructed, and specify the behaviour of each individual component. This process could involve multiple levels of hierarchical structural decomposition and behavioural refinement. The verification step requires showing that the behaviour of the composed components satisfies all the initial requirements of the system.

Given that a designer has identified the required system behaviour and has structurally decomposed the system into components, the problem is then one of how to assign behaviours to the individual components in a way that satisfies the overall system behaviour requirements. Figure 2.3 illustrates three possible approaches to the problem of describing the behaviour of interconnected components. On the left side of the diagram is a system which a designer has structurally subdivided into a set of components. Illustrated on the right side of the diagram are three options available to the designer for describing the behaviour of the system in terms of these components:

- *Resource oriented behaviour specification.* The designer can describe the behaviour of the system by defining behaviour of each individual component. The system behaviour is then given by the composition of the individual component behaviours. If each component is viewed as being a *resource*, this style of behaviour specification is then defined as *resource oriented behaviour specification* since the behavioural description of the system is distributed

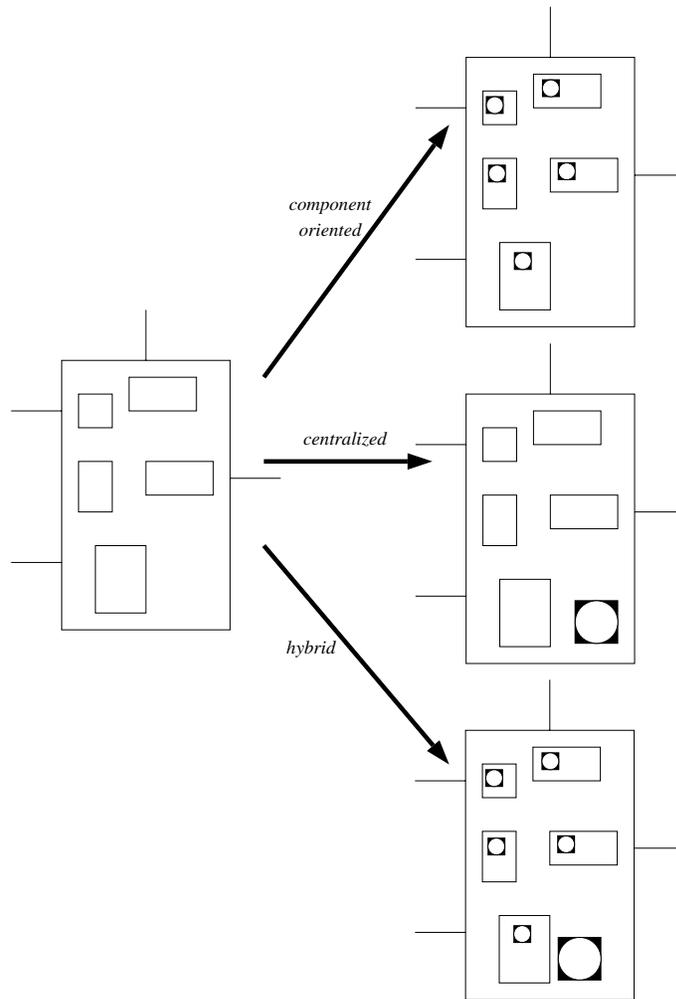


Figure 2.3 Describing behaviour of interconnected components.

among the various components of the system. Examples of resource oriented behaviour specification methods include [5,14,43,20,22,52,77].

- *Centralized behaviour specification.* The designer can describe the behaviour of the system by defining a behaviour which shows how the components interact and cooperate with each other in order to provide the required system behaviour. Such a design perspective views the system as having a centralized controller or monitor; the system behaviour is then described in terms of how the components interact with each other. Note that the use of a centralized monitor to describe behaviour does not imply that the monitor is an *implementation level* artifact, i.e., there will not necessarily exist within the implementation a single entity which implements the central monitor. In this context, the centralized monitor is a *design level* artifact; it describes the required behaviour of the set of components in terms of how they must cooperate with each other, but does not assume that this cooperative behaviour will transform directly into an implementation artifact. Further design steps may be required to distribute the centralized behaviour description design artifact among the various component implementation artifacts. Examples of design methods which use a central monitor to specify behaviour of interconnected components include [13,17,24,38,48,59,63].
- *Hybrid behaviour specification.* Centralized and resource oriented behaviour descriptions are two extremes of a spectrum of behavioural specification approaches. When describing the behaviour of interconnected components there are some behavioural requirements which can be isolated as belonging to a particular component and best described as being within that component, while other behavioural requirements are best described by specifying how sets of components interact with each other to achieve the system functionality. Hybrid behavioural descriptions combine resource oriented and centralized descriptions, allowing each behavioural requirement to be specified in the most appropriate manner. The hybrid approach also permits a certain

amount of redundancy in the behavioural description whereby the centralized behaviour description can be used as a basis for testing the component behaviours developed at a later stage of the design process.

Whether a designer chooses to use a resource oriented approach, or a centralized approach depends on a number of factors such as:

- the most appropriate and natural approach to describing the behaviour.
- the purpose for which the behavioural specification will be used.

Unfortunately sometimes these factors dictate different approaches to behavioural specification. For example, if the purpose of a behavioural design is to provide separate and distinct specifications for each component in order that they can be implemented independent of each other, then a resource oriented behaviour description can be used. However, if the behaviour of the interconnected components requires complex interactions among sets of components it is often easier for the designer to understand the overall behaviour of the system by thinking of behaviour crossing component boundaries. Such an approach is closer to the techniques often found in design methods which focus on overall system requirements rather than individual component behaviours [58].

The design process proposed by this research is illustrated in figure 2.4. A system is constructed from a set of components, as illustrated by the leftmost figure in the diagram. The problem faced by the designer is how to develop the behavioural design of the system in terms of these interconnected components. This can be done initially by a hybrid behaviour description (the middle step of figure 2.4): behavioural requirements which are most appropriately described as belonging to an individual component are described as part of that component's behaviour (e.g., behaviour specifications *b1* through *b5*); behavioural requirements which involve cooperation among many components are defined independent of any individual component and as part of the system behaviour (e.g., behaviour specification *bs*).

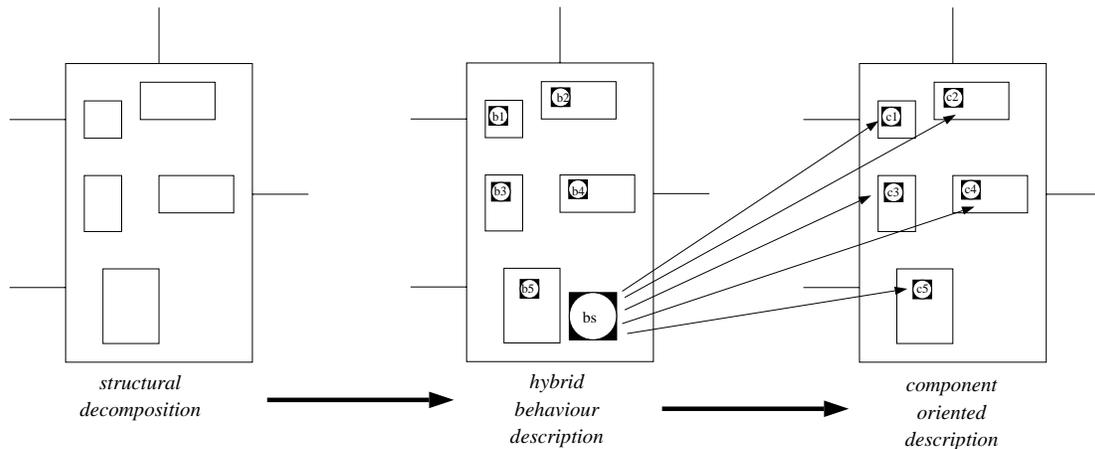


Figure 2.4 The design process: hybrid behaviour description.

Since the behaviour specified within the central behaviour description crosses component boundaries, and the components represent the implementation artifacts, this centralized behaviour specification must be distributed among the components. This is illustrated by the rightmost portion of figure 2.4 where the behaviour bs has been distributed and combined with the behaviours b_1 through b_5 to give the component behaviour expressions c_1 through c_5 . Behaviour which is specified centrally, and not as part of any specific component, is distributed among the individual components and combined with the component behavioural requirements in such a way that the overall system requirements are maintained. If this is done correctly, then the components can be implemented as separate artifacts while still satisfying the behaviour requirements which cross component boundaries. The problem of deriving resource oriented specifications from centralized behaviour specifications will be discussed further in Chapter 5

2.2 Models of Behaviour

This section discusses a number of issues regarding models of behaviour and languages for constructing models of behaviour.

Behavioural design can be represented by constructing a *model* of the system's behaviour. The model is an abstract representation of the intended behaviour of the system which can be used for analysis purposes. The implementation is a concrete realization of the abstract model.

In order to construct a model, a designer uses a behavioural design *language*. The language provides a notation by which a designer can describe the model. The *semantics* of a language define how a specification written in the language is translated into a model. Thus if B is a specification written in some language, then the semantics of the language define the model (or set of models) $\mathfrak{M}(B)$ which correspond to the specification B .

This research is concerned primarily with *event based* models of system behaviour. An event¹ based model defines behaviour by means of identifying temporal relationships between event occurrences. By defining all temporal relationships between event occurrences a complete abstract model of behaviour can be constructed.

In event based models, concurrency can be represented in one of two ways: concurrent events can be represented as being *partially ordered*; or concurrent events can be represented as being *interleaved* in an arbitrary total ordering.

- **Partially Ordered Events** [25,76]: One method of modeling concurrency is to define a relation between event occurrences which is a partial ordering. Events which are unrelated to each other (for example which occur at different concurrent entities) have no temporal relation defined. Two event occurrences which are not related to each other represent actions of the system which may occur simultaneously, be overlapped, or occur in some arbitrary sequential order.
- **Interleaving of Events** [8,46,40,41]: Another method for modeling concurrency is to define a relation between event occurrences which is a total order-

¹ In this thesis an "event" will be defined as an atomic, non-decomposable unit of behaviour.

ing. When a total ordering of event occurrences is used to model a concurrent system, events are assumed to be atomic with respect to each other and are modeled as ‘instantaneous’ actions. Any observer watching an execution of a system and noting event occurrences can put a total ordering on the event occurrences; event occurrences are never simultaneous and when observing two events the observer can always determine which occurrence was first. Concurrency between events is modeled by representing all possible interleavings of the event occurrences.

2.2.1 Behavioural Specification Languages

A behaviour specification model is a means of abstractly representing the behaviour of a system. When creating a behavioral specification model it is not practical to have a designer describe the model explicitly; for example if the behaviour model is based on states and state transitions, then simply identifying all the states and transitions will result in a large incomprehensible specification (particularly if there is an infinite number of states!) A behaviour specification language provides a notation by which a designer can describe the behavioural model (or set of models) in a concise and comprehensible form. Given a specification B , the semantics of the language define the particular model or models $\mathfrak{M}(B)$ which is being represented by the specification.

There are a number of properties of specification languages which are of interest to this research:

- Specification styles.
- Executability.
- Verification techniques.

2.2.1.1 Specification Styles

Behavioural specifications of large complex systems are invariably large and complex. One of the greatest problems in writing specifications for such systems

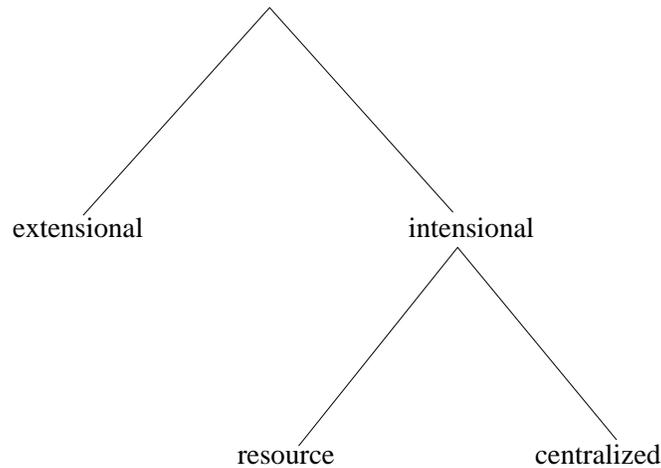


Figure 2.5 Specification styles: extensional and intensional.

is how to make the specification understandable. Presenting an implementor with a thick design document which is not organized and structured will result in an implementor with no understanding of what is to be implemented. Specification languages play an important role in making a specification understandable. What is required in the specification language is a means of using constructs of the language which not only describe the model but also organize the behavioural description in a concise and comprehensible format allowing a reader to understand the high level concepts of the behavioural specification as well as the low level detail.

In order to make a large behavioural specification understandable a number of *styles* of specification have been identified [84,85] as appropriate for different purposes and during different phases of the design process. These styles (modified and extended for this work) can be identified as follows:

- An *extensional* versus an *intensional* style (figure 2.5). A specification is *extensional* if it defines the behaviour of a system entirely in terms of the interactions which occur between the system and its environment without any

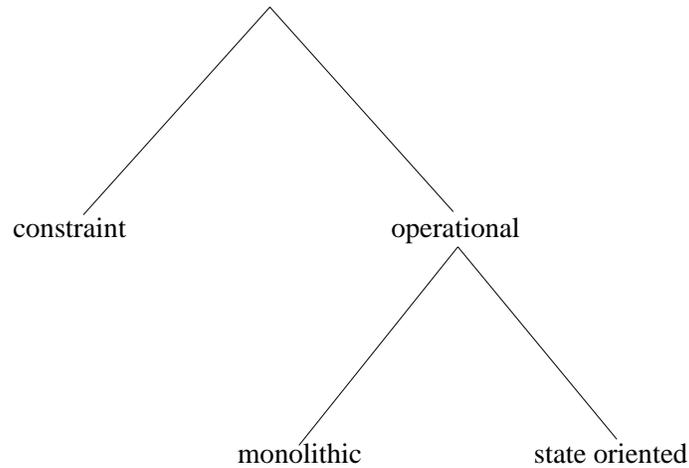


Figure 2.6 Specification styles: operational and constraint oriented.

reference to the internal interactions and structure of the system. A specification is *intensional* if it defines behaviour not only in terms of the interactions between a system and its environment but also uses the interactions which are internal to the system, involving the subcomponents from which the system is constructed. Thus an extensional specification views a system as a black-box while an intensional specification views a system as a white-box.

If an intensional style is being used this can be subdivided further into a centralized or resource oriented behaviour specification (§2.1). In a resource oriented style a separate behavioural specification is constructed for each component; the system specification is then the composition of the individual component specifications. In a centralized style specification, constructs of the language describe behaviour patterns involving many components.

- An *operational* versus a *constraint oriented* style (figure 2.6). In an *operational* style of specification the sequence of operations is stated explicitly as part of an operational specification. Depending on whether a designer wishes to focus on sequences of events or sequences of states, the operational style can be further subdivided into a *state oriented* style or a *monolithic* (i.e., event oriented) style.

A *constraint oriented* style of specification identifies individual constraints which a system must satisfy; the complete specification is then a conjunction of all the individual constraints.

A language which supports all the different styles of specification provides the maximum flexibility for structuring specifications in a comprehensible way and in integrating the language into different phases of the design process.

2.2.1.2 Executable Specifications.

Executable specifications are those which can be executed in order determine the behaviour model represented by the specification [12,43,46,51,89]. Executable specifications have been advocated as a means for improved understanding of the behaviour implied by the specification; if a designer can execute the specification it is easy to determine which behaviour patterns are allowed and which prohibited [88].

A specification S written in language L is executable if there exists an algorithm for constructing the model $\mathfrak{M}(S)$ from the specification S . ‘Executing’ the specification S then consists of using the algorithm (i.e., the operational semantics) and generating the model $\mathfrak{M}(S)$.

For non-executable specifications ([49,1,74,75,79]) there does not exist any algorithmic means of generating the behaviour model $\mathfrak{M}(S)$ from the specification S . In many cases there will not be a single model, but rather a set of models which are all satisfactory models of S .

2.2.1.3 Verification Techniques

Given a formal specification in some language we would like to be able to verify some property of the specification. Verification in this sense of the word means that given a specification S in some language we would like to show that the corresponding model $\mathfrak{M}(S)$ has some property, e.g.:

- Every state of $\mathfrak{M}(S)$ has a successor (absence of deadlock).
- Given an assertion A and a model $\mathfrak{M}(S)$ show that $\mathfrak{M}(S)$ satisfies the assertion A .

There are various methods for verifying properties of a specification. If the specification language is executable then one method is to generate the model $\mathfrak{M}(S)$ and then show that $\mathfrak{M}(S)$ has the required properties [16,51]. Such an approach is known as *reachability analysis* or *model checking*. Clearly complete reachability analysis is not an option for complex systems which have very large or infinite models.

Rather than verifying properties by looking at the model $\mathfrak{M}(S)$, in many systems it is possible to verify properties directly in the specification S without generating the model. This is done by symbolic manipulation of S for example either by having an appropriate derivation system [31,64,66,72,74] or by defining S as an algebra and using standard algebraic techniques [40,41,56,65]. Although such an approach has the advantage that properties about large and infinite size models can be proven, it suffers from the problem in that in general there is no decision procedure which would allow one to automatically verify that specification S satisfies a particular property.

2.3 LOTOS: A Formal Modeling Technique for Behaviour

There currently exist numerous behavioural specification languages with various degrees of formality, executability, models, verification techniques, etc. It is

not the objective of this research to create yet another behavioural specification language and model; rather it is to try to apply a currently existing formalism to the problem of how to develop reliable concurrent systems. The behavioural specification method selected for use in this research is the language LOTOS [46].

LOTOS was initially developed by the ISO for the formal specification of communication protocols [83] but has also been applied to the specification of other types of systems (e.g., [5,23]). It is based on the process algebra CCS [65] for defining event sequences and the data representation language ACT I [19] for representing abstract data types. This section discusses some properties of the LOTOS language. Readers interested in learning the LOTOS language are referred to one of the tutorials [2,61].

The properties of LOTOS which are of interest to this research are the following:

- *Event based.* LOTOS defines the behaviour of a system in terms of event sequences which represent the execution of the system through time. Concurrency is represented by arbitrary interleavings of concurrent events.
- *Executable.* The semantics of LOTOS are defined by means of a set of axiom and inference rules of transition. Given a LOTOS specification, these axioms and inference rules can be applied in order to construct the corresponding transition system model; constructing the model in this way is similar to ‘executing’ a LOTOS specification. Executable specifications have been advocated as a means of facilitating behavioural design [88]; they also permit verification of finite systems to be automated by means of reachability analysis [16,42,54,53].
- *Behaviour expression operators.* Behaviour in LOTOS is specified by writing a *behaviour expression*; a set of algebraic operators provide a mechanism by which a designer can combine two or more behaviour expressions into a new behaviour expression. These operators can assist greatly in combining

behaviour expressions in different ways in order to structure a specification in a comprehensible form. For example:

- If a set of behaviour expressions represent different slice behaviours, these can be combined to show the concurrent execution of the slices.
 - If a set of behaviour expressions represent a set of constraints on system behaviour, these expressions can be combined to represent the conjunction of the constraints.
 - If a set of behaviour expressions represent the behaviours of individual components, these expressions can be composed to represent the interconnected set of components executing concurrently.
- *Different ‘styles’ of behaviour specification*[84,85]. LOTOS can be used to specify behaviour *intensionally* or *extensionally*. As well, a number of different LOTOS ‘styles’ are identified which permit the designer a high degree of flexibility in the approach used to specify behaviour:
 - The ‘constraint’ style represents behaviour as a conjunction of constraints.
 - The ‘state oriented’ style represents behaviour by means of a set of states and state transitions.
 - The ‘monolithic’ style represents behaviour as a choice between sequences of events.
 - *Parameterized behaviour expressions*. If many similar behaviour expressions are required within a single behavioural specification, a single LOTOS *process* can be defined as a parameterized behaviour expression. This process can be invoked with the proper parameters in order to represent the different behaviour expressions.

2.3.1 LOTOS Specification Styles

One criteria which was given as a basis for judging a specification language is how well the constructs of the language can be used to effectively structure the

behavioural specification into a format which can be used and understood during the system's life cycle. A LOTOS specification is structured by constructing *behaviour expressions* and then using the algebraic operators of the language to combine these behaviour expressions into larger expressions. A behaviour expression can be parameterized and represented as a LOTOS *process*¹.

Behaviour expressions and processes are the primary means by which LOTOS can support the different specification styles discussed in §2.2.1.1. Depending on the specification style being used the LOTOS processes within the specification represent different aspects of behaviour. For example:

- Using a resource oriented style, individual processes within the specification represent the behaviour of individual components.
- Using a constraint oriented style, individual processes within the specification represent separate constraints.
- Using a state oriented style, a process can be used to represent a state transition system.
- Using a centralized behaviour specification, a process can be used to represent cooperative behaviour between components.

Some of these styles are illustrated in the following pages. The examples are based on the producer/consumer design structure of figure 2.7² which consists of two producer components (*prod1* and *prod2*) and two consumer components (*cons1* and *cons2*) which communicate by means of a buffer (*buff*). (This example will be reintroduced and described more formally in Chapter 4.) A new element introduced in this diagram is the *interaction point* represented in the diagram as a short, thick line segment. Interaction points are places where components can

¹ The term "process" can cause a great deal of confusion to software engineers. A "LOTOS process" should not be confused with a "software process" (such as an Ada task). Rather a LOTOS process is simply an abstract parameterized representation of "behaviour". A LOTOS process can be used to represent the behaviour of a software process but it is much more general and can also be used in many other ways, such as to represent constraints or assertions.

² Although the visual notation for design structure developed in this thesis is similar to the standard visual notation for Petri nets, the two notations are not related.

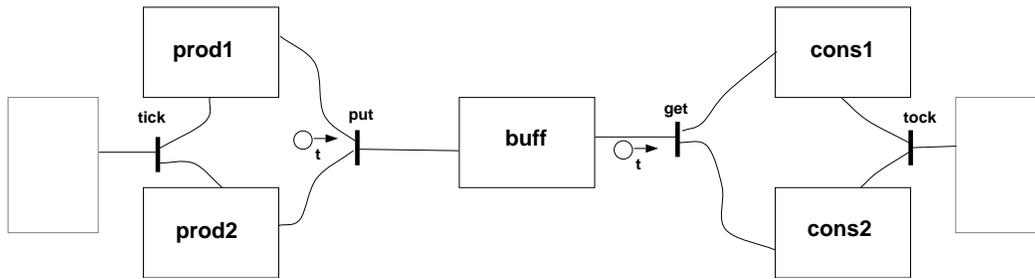


Figure 2.7 Structure of Producer/Consumer system.

synchronize and transfer data values by means of an *interaction*. The transfer of data is indicated by a short arrow with an open circle on the tail. The structural design identifies four interaction points: *tick* by which the producers communicate with the external environment; *tock* by which the consumers communicate with the external environment; *put* where a producer can pass a value of type *t* to the buffer; and *get* where the buffer can pass a value of type *t* to one of the consumers.

Informally, the behaviour of the system can be described as follows:

- Each *tick* interaction will cause one of the producers (*prod1* or *prod2*) to generate a value of type *t* and pass the value to the buffer (*buff*) by means of interaction point *put*.
- Each value received by *buff* will be passed to one of the consumers (*cons1* or *cons2*) by means of the *get* interaction point.
- Whenever a consumer receives a value it will engage in a *tock* interaction. Thus for every *tick* interaction there will be a corresponding *tock* interaction.
- There can be no more than six outstanding *tick* interactions for which the corresponding *tock* interactions have not occurred.

Constraint Style.

To construct a constraint oriented specification a designer identifies the constraints to be specified, expresses each constraint as a LOTOS process, and then

```

1 behaviour
2   { * behaviour of system is defined by conjunction of the constraints
3     const1 and const2 * }
4   const1[tick,tock]
5   ||
6   const2[tick,tock] (0)

7   where

8     { * process const1 defines the constraint: for every tick there is a
9       corresponding tock * }
10    process const1[tick,tock]:noexit :=
11      tick;
12      (const1[tick,tock]
13        |||
14        tock;stop)
15    endproc

16    { * process const2 defines the constraint: there are never
17      more than 6 outstanding ticks. * }
18    process const2[tick,tock] (count:Nat):noexit :=
19      [count lt 7] -> tick;const2[tick,tock] (count+1)
20      []
21      tock;const2[tick,tock] (count-1)
22    endproc

```

Figure 2.8 Example of constraint style in LOTOS.

forms the conjunction of the constraints using the LOTOS algebraic operators. For example assume that two extensional constraints on the producer/consumer design are specified as follows:

1. For every *tick* interaction there is a corresponding *tock*; and a *tock* cannot occur without its preceding *tick*.
2. There can be no more than 6 outstanding *tick* interactions which are waiting for a *tock*.

Each one of these constraints can be represented as an individual LOTOS process. For example constraint 1 is represented as process `const1` in figure 2.8. The process definition is at lines 10 through 15 of the specification. The process definition specifies that a *tick* interaction occurs (line 11) followed by a recursive

invocation of `const1` (line 12) and a *tock* interaction (line 14). The interleaving operator ‘`||`’ at line 13 indicates that the recursive invocation of `const1` and the *tock* interaction occur in parallel.

Constraint 2 is represented as process `const2` defined at lines 18 through 22. The parameter `count` defined at line 18 records the number of outstanding *tick* interactions. The behaviour expression at line 19 states that if `count` is less than 7 then engage in a *tick* interaction and recursively invoke `const2` with the value of `count` incremented. The behaviour expression at line 21 engages in a *tock* interaction and recursively invokes `const2` with the value of `count` decremented. These two expressions are combined using the choice operator ‘`[]`’ at line 20.

Forming the conjunction of these constraints corresponds to combining the process invocations with the synchronizing operator ‘`||`’ (lines 4 to 6). The synchronizing operator ‘`||`’ means that the processes run in parallel but must synchronize on the interactions *tick* and *tock*.

State Oriented Style.

A state oriented style of specification represents behaviour as a state transition system. LOTOS can be used to represent state transition systems in a structured fashion. The usual approach to do this is to represent a state transition system as a process with a value parameter of the process used as a state variable to record the state. Every event occurrence causes a state transition with the process being recursively invoked with an updated state variable. The example for the producer consumer problem is illustrated in figure 2.9. (A visual representation of an equivalent system is shown in figure 2.10.) The state variable in the process definition is called `state` and is declared as a parameter of the process `fsm` at line 3. The initial invocation of the process at line 1 sets the state variable to `state0`.

The system has seven states corresponding to how many outstanding *tick* interactions there are. The state transitions are defined in lines 4 through 26. The choice operators ‘`[]`’ at lines 5, 9, 13, 17, 21, and 25 select between the different

```

behaviour
  { * Behaviour of system is defined by state machine beginning in
    state state0 * }
1  fsm[tick,tock] (state0)

2  where
3    process fsm[tick,tock] (state:State) : noexit :=

      { *If in state0: execute a tick and go to state1 * }
4      [state eq state0] -> (tick;fsm[tick,tock] (state1))
5      []

      { *If in state1: execute a tick and go to state2; or
        execute a tock and go to state0 * }
6      [state eq state1] -> (tick;fsm[tick,tock] (state2))
7                          []
8                          (tock;fsm[tick,tock] (state0))
9      []

      { *If in state2: execute a tick and go to state3; or
        execute a tock and go to state1 * }
10     [state eq state2] -> (tick;fsm[tick,tock] (state3))
11                          []
12                          (tock;fsm[tick,tock] (state1))
13     []

      { *If in state3: execute a tick and go to state4; or
        execute a tock and go to state2 * }
14     [state eq state3] -> (tick;fsm[tick,tock] (state4))
15                          []
16                          (tock;fsm[tick,tock] (state2))
17     []

      { *If in state4: execute a tick and go to state5; or
        execute a tock and go to state3 * }
18     [state eq state4] -> (tick;fsm[tick,tock] (state5))
19                          []
20                          (tock;fsm[tick,tock] (state3))
21     []

      { *If in state5: execute a tick and go to state6; or
        execute a tock and go to state4 * }
22     [state eq state5] -> (tick;fsm[tick,tock] (state6))
23                          []
24                          (tock;fsm[tick,tock] (state4))
25     []

      { *If in state6: execute a tock and go to state5 * }
26     [state eq state6] -> (tock;fsm[tick,tock] (state5))
27     endproc

```

Figure 2.9 Example of state oriented style in LOTOS.

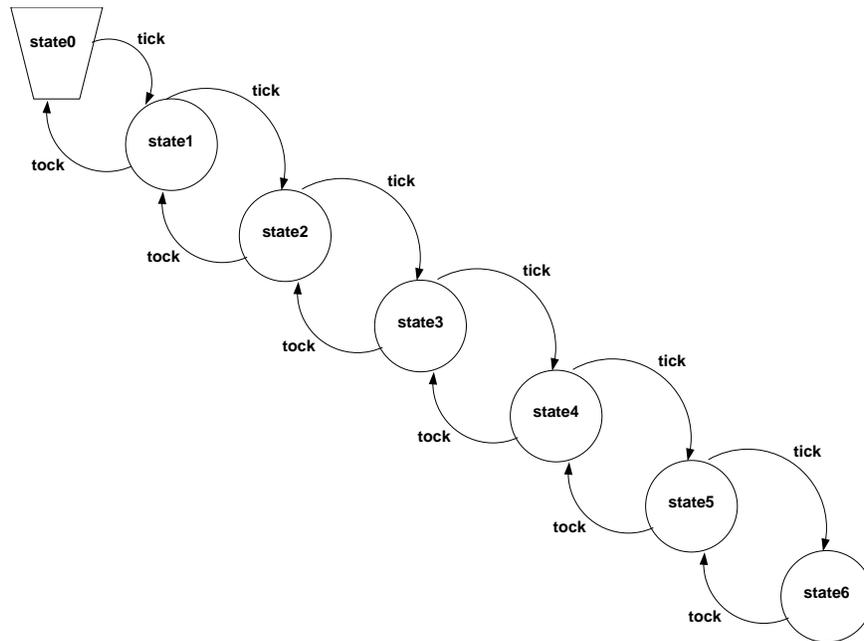


Figure 2.10 Example of state oriented style in LOTOS – visual representation.

choices available depending on the current state. For example, lines 6 through 8 define the behaviour of the system if `state` equals `state1`. In this case the system engages in a *tick* interaction and goes to state `state2` (line 6); or engages in a *tock* interaction and goes to state `state0` (line 8).

Note that specification of figure 2.9 and the specification of figure 2.8 are equivalent in the behaviour which they represent¹.

¹ The two specifications are equivalent in that their semantic models are the same, i.e., $S1$ equivalent to $S2$ since $\mathfrak{M}(S1) = \mathfrak{M}(S2)$ (with a possible renaming of states). This equivalence relation is much stronger than is needed in most cases; a further discussion of relations between models is in §2.3.3.

Intensional Approaches to Specification. An intensional approach to system specification views a system as an interconnected set of components and then defines behaviour not only in terms of how the system interacts with its environment, but also in terms of how the internal components of the system interact with each other. In §2.1.2 two approaches to intensionally specifying a system were defined: a *resource oriented* style where the behaviour of each component is defined separately and the overall behaviour is the composition of these individual specifications; and a *centralized style* where the designer identifies behaviour patterns cutting across many components.

The resource oriented style in LOTOS can be represented by specifying the behaviour of each component as a LOTOS process, and then using the LOTOS algebraic operators to compose these component specifications according to the component interconnection topology. Although this is one approach to using LOTOS in an intensional specification, it suffers from a number of weaknesses (these issues and solutions are discussed further in §4.2.1):

- *No explicit structural information.* LOTOS defines behaviour in terms of event sequences, but does not have any explicit means of representing structural design. Thus the corresponding transition system model contains no structural information. In particular, from the transition system model there is no way to determine which components of the system were involved in a particular interaction. This results in a number of problems, such as the inability to specify intensional behaviour using a centralized approach.
- *Wiring problem.* Using the LOTOS operators, it is not possible to represent an arbitrary network of components which communicate by means of synchronous interactions.

The structure of the producer/consumer example given in figure 2.7 consisted of two producers, two consumers, and a two place buffer. The resource oriented specification is illustrated in figure 2.11 where three processes are defined:

```

1  behaviour
2  hide put,get in
   {*start up two producers and two consumers - no synchronization*}
3  (producer[tick,put]
4  |||
5  producer[tick,put]
6  |||
7  consumer[get,tock]
8  |||
9  consumer[get,tock])

   {*Buffer synchronizes with producers and consumers on put and get*}
10 | [put,get] |
11  buffer[put,get]

12  where
   {*define the behaviour of a producer*}
13  process producer[tick,put]:noexit :=
14  choice N:Nat []
15  tick;
16  put !N;
17  producer[tick,put]
18  endproc

   {*define the behaviour of a consumer*}
19  process consumer[get,tock]:noexit :=
20  get ?N:Nat;
21  tock;
22  consumer[get,tock]
23  endproc

   {*Define the behaviour of a buffer*}
24  process buffer[put,get]
   {*Two place buffer can be constructed as two one place buffers
25  executing in parallel*}
26  (buff[put,get]
27  |||
28  buff[put,get])

29  where
30  process buff[put,get]:noexit :=
31  put ?N:Nat;
32  get !N;
33  buff[put,get]
34  endproc
35  endproc

```

Figure 2.11 Example of resource oriented style in LOTOS.

- `producer` which represents the behaviour of a generic producer (lines 13 through 18).
- `consumer` which represents the behaviour of a generic consumer (lines 19 through 23).
- `buffer` which is the behaviour of a two place buffer (lines 24 through 35).

The behaviour expression is then created by instantiating the `producer` and `consumer` processes twice to represent the producers and consumers (lines 3, 5, 7, and 9), and instantiating the `buffer` process (line 11). Since the producers and consumers do not interact directly with each other they are combined using the full parallel operator ‘`|||`’ (lines 4, 6, and 8). The buffer communicates with the producers and consumers using the interaction points `put` and `get`; this is reflected by having the producers and consumers synchronize with the buffer on these two interaction points using the operator ‘`| [put, get] |`’ (line 10).

Since the interactions `put` and `get` are internal and do not involve any external components, the `hide` operator is used so that they are not visible to the environment (line 2).

Note how the top level behavioural expression of figure 2.11 mirrors the structural specification of figure 2.7, with a unique process invocation used to represent the behaviour of each component, and these processes combined together in a way which corresponds to the component interconnection topology defined in the design structure.

2.3.2 Verification in LOTOS

A verification of a LOTOS specification S can be done in a number of ways. First, since LOTOS is an executable language it is possible to generate the model $\mathfrak{M}(S)$ and then to verify that $\mathfrak{M}(S)$ satisfies the required properties. As with any reachability analysis this can be done only if the number of states and number of transitions in the model is finite and tractable.

A second method of verification takes advantage of the formal algebra which forms the basis for LOTOS. Such a verification generally follows the format of looking at two different specifications which represent different views of the problem and then proving by term rewriting that some relation exists between these two different specifications. For example if we look at the producer/consumer example whose structure was given in figure 2.7, a number of different behavioural specifications were provided for this structure including:

- An extensional constraint oriented specification (figure 2.8) which defined the behaviour of the system in terms of the conjunction of a set of constraints on the interactions between the system and its environment.
- An intensional resource oriented specification (figure 2.11) which defined the behaviour of the system as a set of interconnected components.

These two specifications are different both in their specification styles and in their structural perspectives; however they are both intended to represent the behaviour of the same system. A verification would determine whether the resource oriented intensional specification is equivalent to the constraint oriented extensional specification. Such a verification can be done entirely by symbolic manipulation of the specifications without generating the corresponding models.

Although in theory it is possible to perform such verifications, in practice it is extremely difficult due to the large and complex nature of most of the proofs. With no automated way of constructing such proofs, verification is a time consuming, error-prone operation requiring highly skilled people.

For LOTOS the problem of verification by proving equivalence between two behaviour specifications is compounded by having two different algebras included within a single formal language. For many specifications it is not clear what should be specified using the 'process algebra' and what should be specified using the 'data algebra'. Trying to verify relations between specifications which divide a

problem differently according to process and data is an area which has not been fully explored [60].

2.3.3 The Transition System Model

LOTOS models behaviour in terms of events, where an event is an atomic, non-decomposable, instantaneous occurrence. At most one event can be occurring at any instant in time; thus concurrency of a set of events is modeled by all possible interleavings of the concurrent events. This leads to a model of behaviour which is represented as a transition system:

- Each node of the transition system represents a state of the system.
- Each edge of the transition system represents an event occurrence causing a state transition. The special event i is used to represent a ‘hidden’ event which cannot be observed.

A valid execution of the system is any path originating at the initial state.

Events being atomic are non-decomposable; as a specification is decomposed the ‘states’ can be decomposed into substates and subsequences of events, but the events at one level must remain as events during the decomposition.

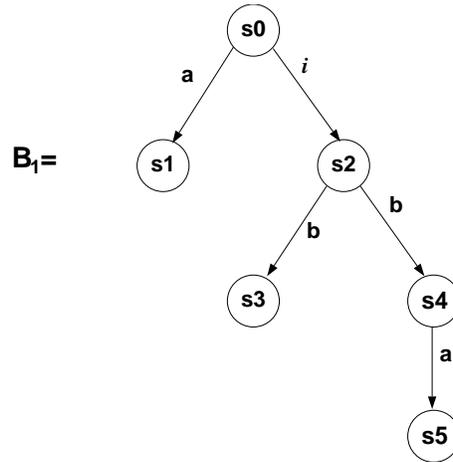
Formally, the LOTOS model is defined as a *Transition System (TS)*.

Definition: Transition System (TS)

A *Transition System* is a tuple $\langle \Sigma, I, T, s_0 \rangle$ where:

- Σ is a countable set of states.
- I is a set of events.
- T is a set of transitions defined as a set of binary relations on Σ such that if $\xrightarrow{e} \in T$ then $e \in I \cup \{i\}$
- $s_0 \in \Sigma$ is the initial state.

Figure 2.12 illustrates a transition system called B_1 .



$$\Sigma = \{s_0, s_1, s_2, s_3, s_4, s_5\}$$

$$I = \{a, b\}$$

$$T = \left\{ s_0 \xrightarrow{a} s_1, s_0 \xrightarrow{i} s_2, s_2 \xrightarrow{b} s_3, s_2 \xrightarrow{b} s_4, s_4 \xrightarrow{a} s_5, \right\}$$

Figure 2.12 Example transition system B_1 .

The set Σ represents all possible states of the system where the *state* encapsulates all possible future behaviour of system. In the definition of the LOTOS semantics, each state of the system is identified by means of a LOTOS behaviour expression.

The set I represents all possible observable events which can occur. For each interaction e there is a relation $\xrightarrow{e} \in T$ which defines the state transition which results from the interaction occurring. If $(s_1, s_2) \in \xrightarrow{e}$ this is normally written as $s_1 \xrightarrow{e} s_2$. In addition to the events of I there is an *unobservable event* denoted as i which also has associated with it a state transition relation. The event i represents an unobservable interaction occurring within the system.

Every LOTOS specification represents a transition system model. Every transition system contains a set of observable events, and an unobservable event

denoted by i . When studying properties of transition systems, we are interested only in the observable behaviour of a system. (The meaning of ‘observable behaviour’ in the context of a concurrent set of components will be discussed further in §4.2.1.1.)

One method of characterizing the observable behaviour of a transition system is by the sequences of observable events in which it can engage. For a transition system B , this set of observable sequences is defined as $traces(B)$.

For a transition system B which contains nondeterminism the set $traces(B)$ is not sufficient for completely characterizing the observable behaviour of B (see §4.2.1.1 and §5.1.1). In order to characterize the observable behaviour of a nondeterministic transition system B , we must be able to observe not only the sequences of observable events which occur, but also the events in which B *refuses* to engage¹. This leads to the concept of a *failure* of B (see Brookes et. al. [8]). The pair $\langle t, A \rangle$ is an element of $failures(B)$ if and only if t is in $traces(B)$ and after executing trace t the transition system B can refuse to engage in any event of the event set A .

The set of *failures* provides a means of characterizing the observable behaviour of a transition system. It also provides a means of defining some important relations between transition systems. This research is particularly concerned with incremental development of behavioural design, whereby a designer begins with simple behaviour patterns and then extends these simple behaviours into more complex behaviours. Thus if a designer begins with behavioural specification B_1 , and then refines this into specification B_2 , we wish to define a relation between B_1 and B_2 which captures the concept that B_2 contains at least as much behaviour

¹ This is sometimes referred to as the “vending machine” model. Imagine a vending machine in which the user inserts coins and then pushes a sequence of buttons to make a selection. Depending on the state of the machine (e.g., how much money has been inserted), only a subset of the selection buttons are enabled at any one time. Each selection button has a light. If a selection button is enabled and that selection can be made then the light is on; else the light is off. A user of the system can observe the coins being inserted and the selection buttons being pushed, i.e., the *traces* of the system. The user can also observe which buttons are being *offered* and which are being *refused* by noting which selection button lights are on or off.

as B_1 but may contain more behaviour. This relation is defined as ‘*ext*’ and can be stated as $B_2 \text{ ext } B_1$.

The remainder of this section provides formal definitions of the sets *traces* and *failures*, and the relation *ext*. Similar presentations of the material can be found in many other sources, including for example [7,6].

For transition system $\langle \Sigma, I, T, s_0 \rangle$, every event $e \in I \cup \{i\}$ defines a transition $\xrightarrow{e} \in T$. Each state transition of the transition system is then represented as $s_1 \xrightarrow{e} s_2$. A sequence of state transitions $s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \dots \xrightarrow{e_n} s_{n+1}$ with the e_i in $I \cup \{i\}$ will be represented as: $s_1 \xrightarrow{e_1.e_2..e_n} s_{n+1}$. In the example transition system B_1 of figure 2.13 $s_0 \xrightarrow{i.b.a} s_5$. For all states s , $s \xrightarrow{\lambda} s$ where λ is the null sequence of events.

Given that we are interested in sequences of *observable* events, a notation can be defined for representing sequences with the i events removed. Thus if:

$$s_1 \xrightarrow{i^{m_1}.e_1.i^{m_2}.e_2..i^{m_n}.e_n.i^{m_{n+1}}} s_{n+1}$$

with the e_i in I then this is represented as

$$s_1 \xrightarrow{e_1.e_2..e_n} s_{n+1}$$

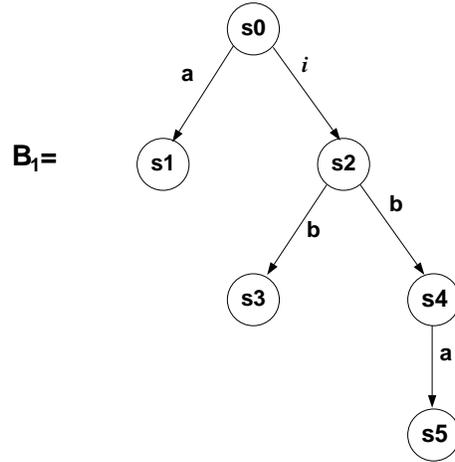
In the example transition system B_1 , $s_0 \xrightarrow{b.a} s_5$. For all states s , $s \xrightarrow{\lambda} s$.

Given behaviour B which represents a transition system $\langle \Sigma, I, T, s_0 \rangle$, we can define the *traces* of a state $s \in \Sigma$ as the set of all visible sequences of events which can occur from state s . The empty trace λ is a trace of every state of Σ . In the example transition system B_1 the set $\text{traces}(s_0) = \{\lambda, a, b, b.a\}$.

Definition: traces(s)

$$\text{traces}(s) \stackrel{\text{def}}{=} \left\{ t \in I^* \mid \exists s_n \in \Sigma \bullet s \xrightarrow{t} s_n \right\}$$

For a transition system B the set $\text{traces}(B)$ is defined as $\text{traces}(s_0)$.



$$\Sigma = \{s_0, s_1, s_2, s_3, s_4, s_5\}$$

$$I = \{a, b\}$$

$$T = \left\{ s_0 \xrightarrow{a} s_1, s_0 \xrightarrow{i} s_2, s_2 \xrightarrow{b} s_3, s_2 \xrightarrow{b} s_4, s_4 \xrightarrow{a} s_5, \right\}$$

$$traces(B_1) = \{\lambda, a, b, b.a\}$$

$$after(B_1, b) = \{s_3, s_4\}$$

$$refusals(s_0) = \{\emptyset, \{a\}\}$$

$$failures(B_1) = \{ \langle \lambda, \emptyset \rangle, \langle \lambda, \{a\} \rangle,$$

$$\langle a, \emptyset \rangle, \langle a, \{a\} \rangle, \langle a, \{b\} \rangle, \langle a, \{a, b\} \rangle,$$

$$\langle b, \emptyset \rangle, \langle b, \{a\} \rangle, \langle b, \{b\} \rangle, \langle b, \{a, b\} \rangle,$$

$$\langle b.a, \emptyset \rangle, \langle b.a, \{a\} \rangle, \langle b.a, \{b\} \rangle, \langle b.a, \{a, b\} \rangle \}$$

Figure 2.13 Example transition system.

The transition systems of interest need not be deterministic; thus given a trace t of s there is not necessarily a unique state s_n such that $s \xrightarrow{t} s_n$. There may be many such states reachable from s by means of trace t . We can therefore define a set of states which are reachable from s after trace t . In the example transition system B_1 the set $after(s_0, b) = \{s_3, s_4\}$.

Definition: after(s,t)

$$\text{if } t \in \text{traces}(s) \text{ then } after(s,t) \stackrel{def}{=} \left\{ s' \in \Sigma \mid s \xrightarrow{t} s' \right\}$$

The set $after(B,t)$ is defined as $after(s_0,t)$.

For each state $s \in \Sigma$ we can define a *refusal* of s as being the set of observable events in which s may refuse to engage. Some care is required to precisely define what is meant by ‘refusing an observable event’ considering that any number of unobservable events can occur before the refusal. What is meant is the following:

- s can refuse event e if s can transform to s' by means of a number of hidden events, and s' can refuse to engage in event e after some sequence of hidden events.

A *refusal* of s is then a set of events which may all be refused by s after transforming to s' by a sequence of hidden events. In the example transition system B_1 the set $refusals(s_0) = \{\emptyset, \{a\}\}$. Formally this can be defined as follows.

Definition: refusal

$$refusals(s) \stackrel{def}{=} \left\{ A \in \mathcal{P}(I) \mid \exists s' \in after(s, \lambda) \bullet \forall e \in A \bullet e \notin traces(s') \right\}$$

For a given transition system B , the observable behaviour of B can be characterized by identifying all the *traces* of B and then identifying all the event

sets which may be *refused* by B after trace t . This is defined as the *failures* of B .

Definition: failures(B)

$$\begin{aligned} failures(B) &\stackrel{def}{=} \\ &\{ \langle t, A \rangle \in traces(B) \times \mathcal{P}(I) \mid \exists s' \in after(s_0, t) \bullet \forall e \in A \bullet e \in refusals(s') \} \end{aligned}$$

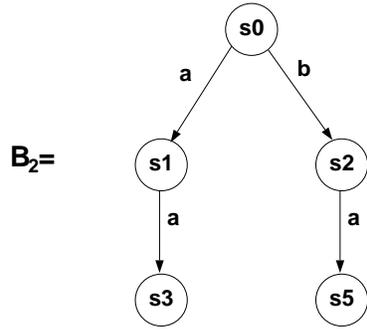
The set of *failures* of a transition system can be used as a semantic model, since any two transition systems with the same *failures* cannot be differentiated by any test performed by an observer [18]. *Traces* and *failures* also provide a basis for the formal definition of the relation *ext*.

Definition: ext

A transition system B_2 *ext* B_1 if and only if:

1. $traces(B_2) \supseteq traces(B_1)$
2. $\forall t \in traces(B_1) \bullet \forall A \in \mathcal{P}(I) \bullet$
 if $\langle t, A \rangle \in failures(B_2)$
 then $\langle t, A \rangle \in failures(B_1)$

Figure 2.14 defines transition system B_2 . By comparing the *traces* and *failures* of B_2 to the traces and failures of B_1 (figure 2.13) it can be verified that B_2 *ext* B_1 .



$$traces(B_2) = \{\lambda, a, b, a.a, b.a\}$$

$$\begin{aligned}
 failures(B_2) = & \{ \langle \lambda, \emptyset \rangle, \\
 & \langle a, \emptyset \rangle, \langle a, \{b\} \rangle, \\
 & \langle b, \emptyset \rangle, \langle b, \{b\} \rangle, \\
 & \langle a.a, \emptyset \rangle, \langle a.a, \{a\} \rangle, \langle a.a, \{b\} \rangle, \langle a.a, \{a, b\} \rangle, \\
 & \langle b.a, \emptyset \rangle, \langle b.a, \{a\} \rangle, \langle b.a, \{b\} \rangle, \langle b.a, \{a, b\} \rangle \}
 \end{aligned}$$

Figure 2.14 Transition system B_2 is an *extension* of B_1 .

Chapter 3: The Design Problem: Behavioural Design by Slicing

When confronted with the problem of specifying the behaviour of concurrent systems, designers often begin by thinking in terms of critical end-to-end behaviour patterns which traverse a set of components. Using such an approach, a designer specifies how a stimulus to the system initiates a behaviour pattern which ripples through the components of the system and results in the system providing some response to the environment. This process of specifying end-to-end behaviour patterns which ‘slice across’ the components of a concurrent system is illustrated in figure 3.1. On the left side of the diagram is a system which has been decomposed into four components. The designer has identified how these four components must interact with each other to achieve some goal, and this behaviour has been superimposed on the structure as slice expression s_1 on the right side of the diagram. The slice s_1 represents an initial stimulus occurring at the left side of the system, and a resulting behaviour pattern which sequences through the components, with a response being output at the top of the system.

System *slicing* is an approach to concurrent system design which specifies behaviour in a centralized style while recognizing that the centralized description is a design artifact and does not transform directly into an implementation artifact. The motivation for using slices is to provide an approach to concurrent system design which has the following characteristics:

- Using slice expressions, a designer can identify major behaviour patterns which flow through the concurrent components of the system and capture these patterns as incomplete behavioural specifications at different stages of the design process.
- The behavioural design problem can be factored in a way which treats key issues first. The designer can identify critical paths through the components

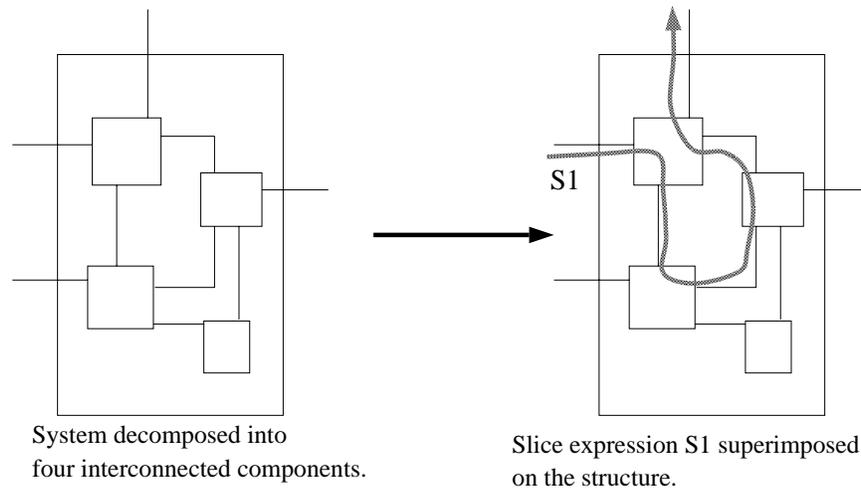


Figure 3.1 Superimposing a slice expression on a structure.

of the system and specify independently the behaviour required along each of these paths.

- Slice specifications permit incremental development of behavioural specifications by allowing the designer to concentrate on behaviour patterns which satisfy the basic requirements of the system, and then to refine these behaviour patterns to provide a more complete specification.
- Slices provide a means for developing and testing component specifications. Once a number of slice expressions have been developed, a designer can use these to determine the behavioural requirements of an individual component, thereby providing a component specification which can be used as a basis for implementing the component. As well, each slice can be used as a test description for testing an individual or a set of component specifications.

3.1 Steps in the Design Process

This thesis is concerned with developing a set of techniques for using slices which can be integrated into a design process. Many possible design processes

might be based on the techniques of this thesis and it is outside the scope of this thesis to develop a specific design process. However, in order to put the techniques developed in this thesis into context, a design process which includes the following steps will be assumed:

1. *Perform the structural design.* The structural design involves identifying the components from which the system is constructed and their interconnection.
2. *Explore the behaviour of the structural design through slice expressions.* Viewing the system as a black box, a designer can define example behaviour patterns showing how the system must interact with its environment. From these patterns, a designer can synthesize a set of components¹ and specify a slice expression to show how the components interact in order to provide the required behaviour pattern. Slices can be used as a means of experimenting with different component decompositions and allocations of functions to components. Beginning with simple slices which show basic behaviour, the designer refines these slices in order to specify more complex behaviour patterns.
3. *Develop component behaviour expressions.* Slice expressions can be used to gain an understanding of intercomponent behaviour; from slice expressions the designer can then develop behaviour specifications for each individual component. This is done by extracting component behaviour from the slice expressions and using this as a basis for developing the specifications of the individual components.
4. *Verify the component behaviour specifications.* Once the complete component specifications have been developed it is necessary to verify that these component specifications satisfy all the requirements implied by the slice expressions.

¹ The specific techniques which a designer uses to determine what are the internal components of a system are not directly addressed by this research.

5. *Decompose complex components.* The structural components can be decomposed into smaller subcomponents and the design process repeated on the subcomponents.

Note that these steps are not necessarily sequential in nature; the actual design process could require a significant amount of backtracking through the steps. The following sections describe these design steps in more detail.

3.1.1 Structural Design

In order to use a slice style of specification it is necessary to separate the structural and behavioural design representations. The examples provided in Chapter 2 did not explicitly separate out these two design concerns.

Structural design involves identification of the components from which a system is constructed, the interconnection mechanism between the components, and the interconnection topology of the components. The structural design step requires a technique for identifying the components and interactions within a system, and a technique by which the structural design can be captured and represented. A method for structural design representation is developed in §4.1. During structural design, slices are useful as a means of exploring design options regarding component decomposition and allocating behaviour to components.

3.1.2 Exploring Behavioural Design with Slices

An individual slice expression is not necessarily intended as a complete specification of behaviour. A designer begins by identifying major behaviour patterns moving through the components and then refines these descriptions in order to represent more elaborate behaviour. This thesis proposes and formalizes two methods for making such refinements (these methods are illustrated using LOTOS in the example of §6.2):

1. *Slice Extension*. A slice specifying one behaviour pattern can be *extended* to illustrate more complex behaviour¹.
2. *Slice Combination*. Two or more slice expressions can be combined to show behaviour patterns operating concurrently within the system.

Refinement by Slice Extension.

Extending slice expressions is a process involving stepwise refinement (figure 3.2). A designer begins with a slice expression s_1 which identifies a particular behaviour of the system. In the simplest case this is represented as a sequence of interactions. Once the specifier is satisfied that s_1 is correct, it can be extended to show possible variations of the behaviour (slice s_1'). The slice s_1' can then be extended with more complex behaviour (slice s_1''). Each slice extension shows different choices and outputs which can result from the initial triggering events. For example, in specifying the behaviour of a distributed telephony system, a designer can begin by showing a simple call connection between two users of the system. The designer can then refine this simple call connection behaviour to illustrate different failure conditions (e.g., destination user not available) or different services which may be provided by the system (e.g., call forward). The example of Chapter 6 will illustrate the concept of refinement by extension using a telephony example.

The actual technique used to refine slices by extension can be divided into two steps: first, the slice is *segmented*; then the new slice is *constructed* from the different segments. These steps are illustrated in figures 3.3 and 3.4 where the slice s_1 is extended into slice s_1' . Given a slice expression s_1 , segmentation identifies sequences of component interactions representing a single phase of the system behaviour. For example, from the slice s_1 two segments have been identified (figure 3.3): s_{1_a} and s_{1_b} ; in addition a third segment s_{1_c} has been defined.

¹ Note that the term *extension* as used in this research has two different meanings depending on the context in which the word is used. Specifically, *slice extension* is a technique for refining slice expressions; the relation *ext* (extension) is a relation between transition system models.

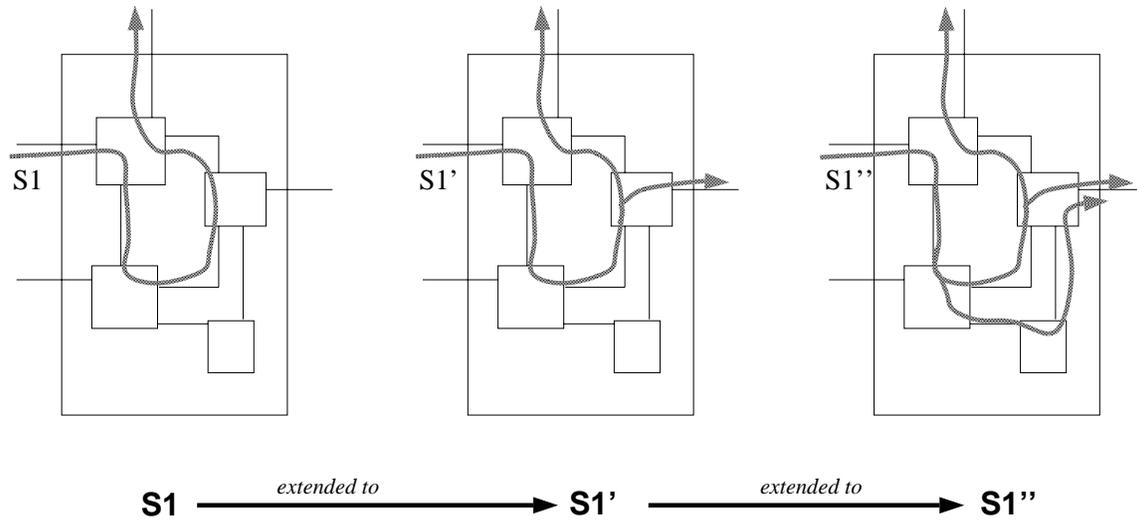


Figure 3.2 Extending simple slice expressions into more complex slice expressions.

Figure 3.4 shows a new slice $s1'$ being constructed from the segments. It defines a behaviour pattern in which the behaviour $s1_a$ is followed either by behaviour $s1_b$ or $s1_c$.

Visual notations which will be used to represent slices and perform the operations of segmentation and construction will be developed in §4.2.2. Refinement by extension is illustrated in the designs presented in Chapter 6, Appendix D, and the case study of [26].

Refinement by Combination.

In addition to constructing more complex slice expressions by extending existing slice expressions, it is possible to construct complex slice expressions by combining existing slice expressions to represent the concurrency between slices. This is illustrated in figure 3.5 where a designer has constructed two slice expressions $s1$ and $s2$ describing different aspects of behaviour. These slice expressions can now be combined in order to specify the concurrent behaviour represented by the two slices; this is represented by the slice expression $s1\#s2$.

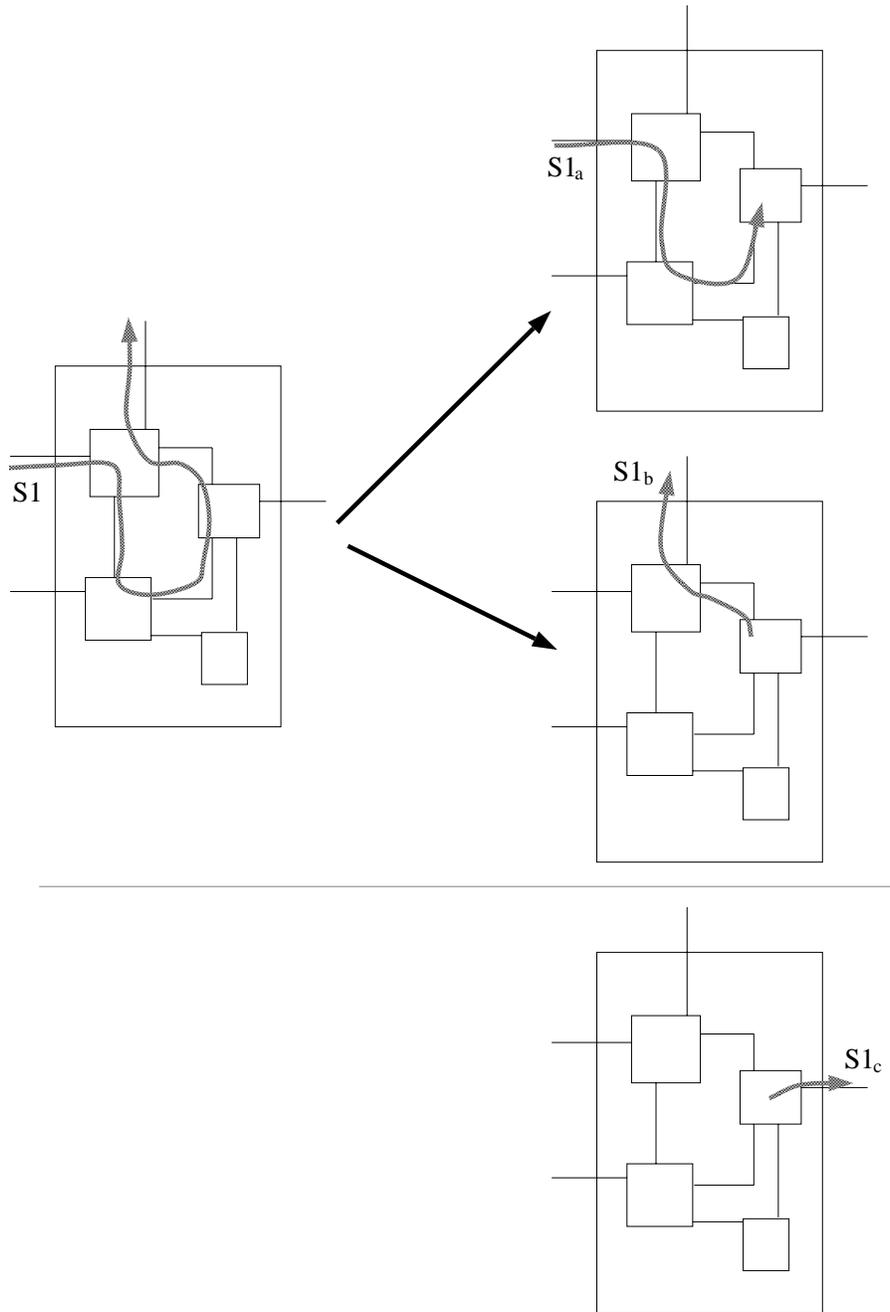


Figure 3.3 Defining slice segments.

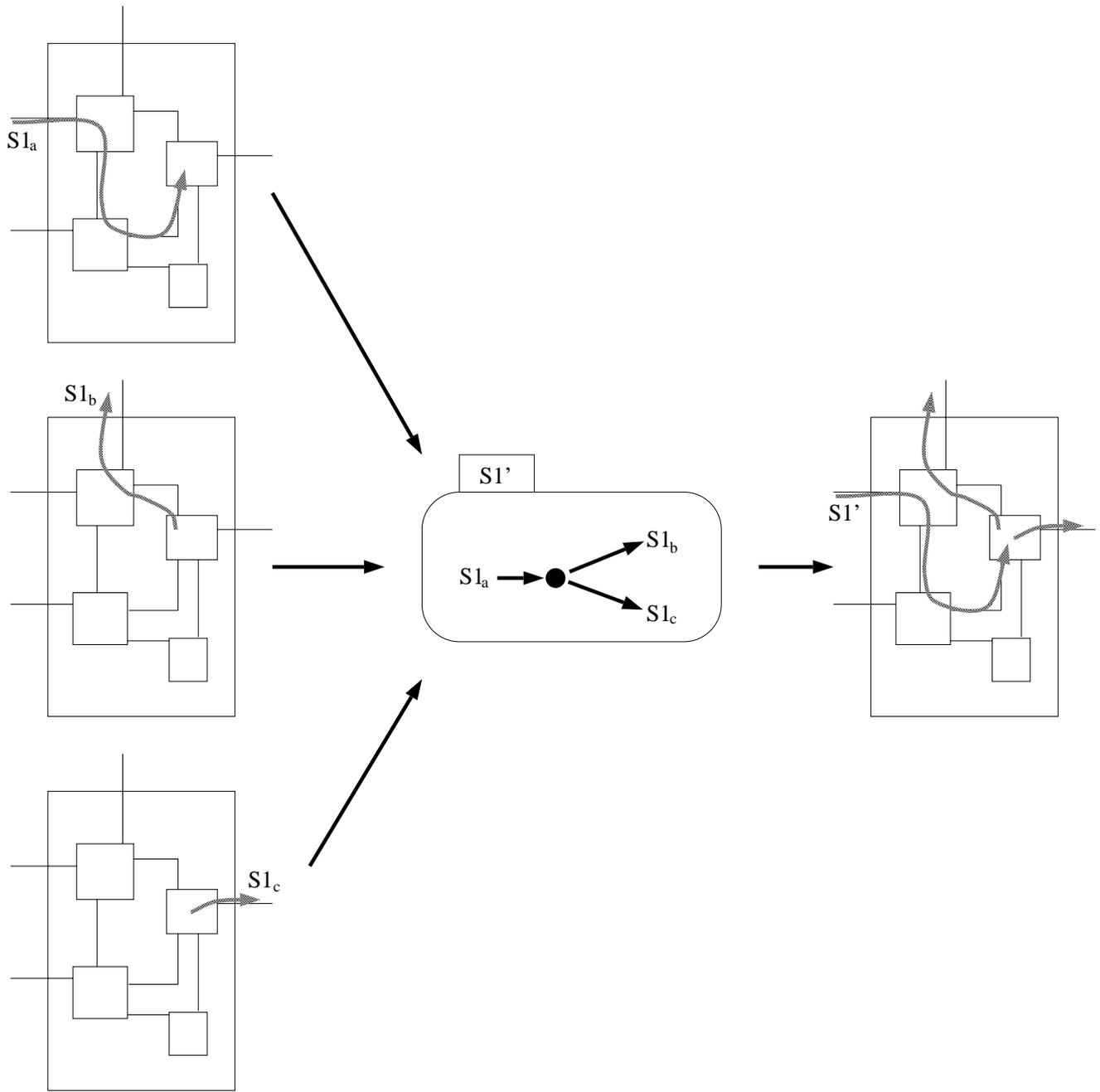


Figure 3.4 Constructing the slice $S1'$ from segments $S1_a$, $S1_b$ and $S1_c$.

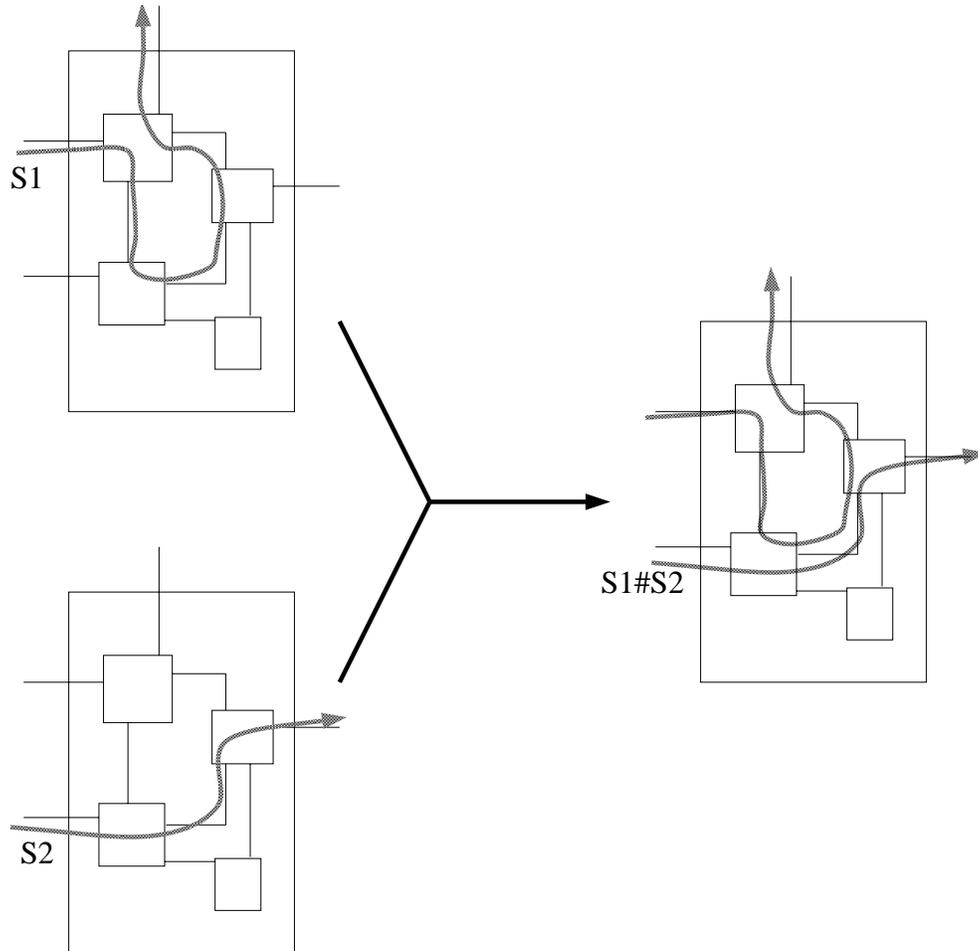


Figure 3.5 Combining slice expressions to create a new slice expression.

The difficulty in specifying the concurrent combination of two slices is that slices are not necessarily independent of each other. Slices may have an effect on each other as they cross over and are intertwined in time and space. A problem arises as to how to represent the interrelationship between the slices.

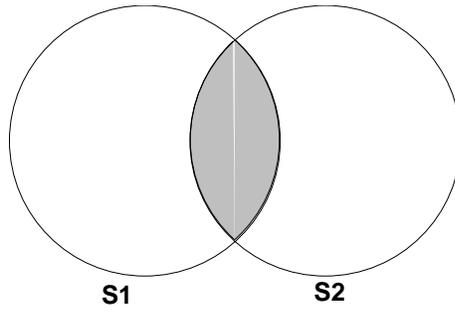
The general approach which is used by this research is illustrated in figure 3.6; these steps are formalized in Chapter 6. The method for combining slices to represent concurrent behaviour can be summarized as follows:

1. *Identify where slice expressions affect each other.* As slice expressions cross over and move through the same components, they may have a behavioural effect on each other. The first step in the combination process is to identify the behaviours of the slices which have an effect on each other.
2. *Divide slices into independent parts.* The original slices s_1 and s_2 can be divided into three parts: the parts s_1' and s_2' which are independent of each other; and the part $s_{(1,2)}$ where the two slices affect each other.
3. *Combine and synchronize the slices.* An operation ‘#’ derived from $s_{(1,2)}$ can be defined to combine and synchronize the slices s_1' and s_2' .

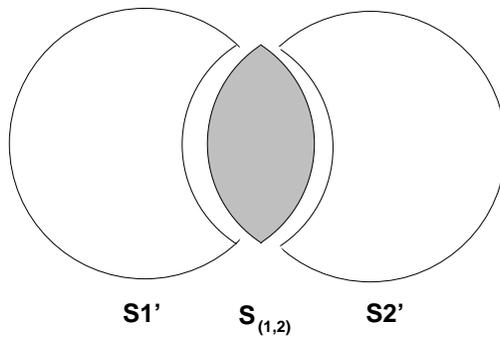
In this research, the synchronization operation ‘#’ is defined by means of a set of behavioural constraints represented as LOTOS expressions (*synchronizing processes*) and combined using the LOTOS operators. Synchronizing processes are defined visually as a state transition system (§4.2.2). Refinement of slice behaviour by combination is illustrated in the designs presented in Chapter 6, Appendix D, and the case study of [26].

3.1.3 Developing Component Specifications

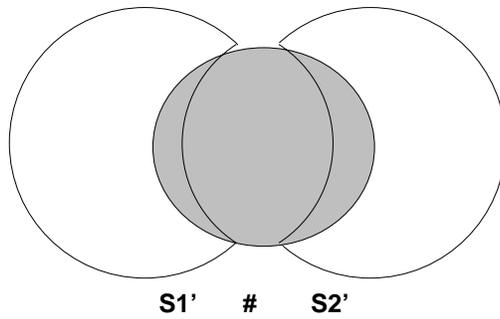
Each slice expression generated by the designer captures some end-to-end functionality which must be exhibited by the set of components from which the concurrent system is constructed. These slice expressions, although not necessarily a complete behavioural specification, form a basis from which the components can be designed and specified.



1. Identify where slices affect each other.



2. Divide slices into independent parts..



3. Synchronize and combine expressions.

Figure 3.6 Process for combining slices.

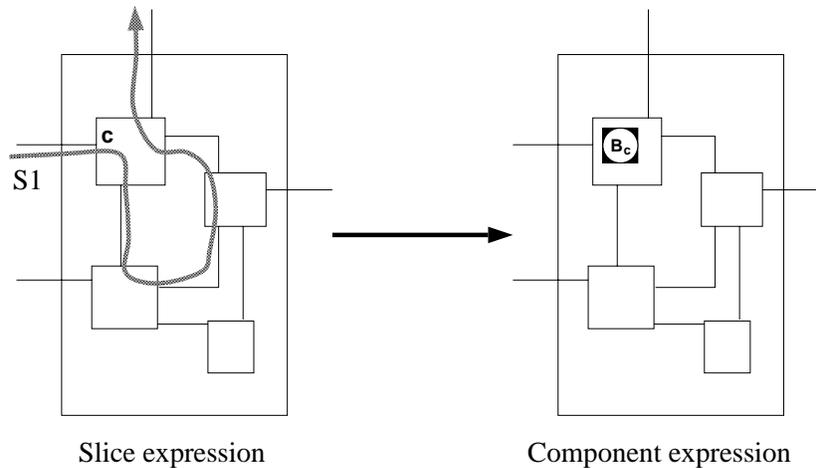


Figure 3.7 Extracting component behaviour from a slice expression.

A slice expression specifies behaviour in a distributed manner. It represents a design artifact, not an implementation artifact. Implementation of the components requires that the behaviour expressed in each slice expression be distributed through the components of the system. The problem is illustrated in figure 3.7. The designer has identified component c of the system. As part of the behavioural design, a slice s_1 has been specified. Using slice s_1 a designer can construct the behavioural specification B_c which describes the behaviour of c and satisfies all requirements implied by slice s_1 . Methods for extracting component specifications from slice expressions are developed in §5.2 and used in Chapter 6, Appendix D, and [26].

Once a set of component specifications have been derived from a slice expression, these component expressions can be composed into a resource oriented specification of behaviour. By comparing the original slice expression to the derived resource oriented expressions it is possible to analyze the design for potential concurrent races. The method of analysis is described in §5.3.

3.1.4 Verifying Component Specifications Using Slice Expressions

Each slice expression captures an aspect of the behaviour which the designer expects to see within the concurrent system. Thus, once the component behaviours have been specified and a resource oriented specification of the system constructed, it can be verified that the component specifications satisfy the requirements imposed by the slice specification. A basis for using slice expressions to verify component specifications is developed in §5.4 and illustrated by example in Chapter 6, Appendix D, and [26].

3.1.5 Structural Decomposition

The design process involves decomposing complex components into networks of simpler components. This structural decomposition is recursively applied until the primitive components of the system are identified. Design issues involving structural decomposition are discussed in §4.1.1.2 and §4.2.3 and applied in the examples of Chapter 6, Appendix D, and [26].

3.2 Related Research

The use of ‘slice behaviour’ as an approach to specifying the behaviour of an interconnected set of components is an active area of research within object oriented design methodologies as applied to sequential (i.e., nonconcurrent) programs. Examples of such an approach for sequential programs include *collaborations*[87], *contracts*[38] and *mechanisms*[4].

Less research work has been done in applying the slice style to concurrent programs. The approach has been advocated by Buhr in [10,9,11]. In Buhr’s work, slices are used either to describe the passing of abstract events between components, or to describe sequences of extended rendezvous between components. Event timelines can be applied at any level of the structural design hierarchy. The research of this thesis extends these ideas primarily by providing a

more formal and rigorous framework for the concept of slicing. Once this formal framework is provided, it is possible to investigate areas such as extracting component behaviour from slice expressions, using slice expressions as a basis for test generation, and providing tool support for these activities.

Describing the behaviour of sets of concurrent components is also used within the DOCASE development environment [67,68]. DOCASE is an object oriented design environment for large distributed systems. Within DOCASE, behaviour patterns involving many concurrent components are defined as *cooperations*. The major differences between the approach taken by DOCASE and the slices of this research include:

- In DOCASE, cooperations are objects which transform directly into an implementation artifact. Thus a cooperation acts as a single controller orchestrating the behaviour of the other objects involved in the behaviour pattern. Slice expressions on the other hand are design artifacts, not implementation artifacts. Although they define behaviour of sets of components, within the implementation the behaviour which implements the slice is distributed among the components involved in the slice expression.
- Within DOCASE the behaviour of cooperations is defined using a conventional object oriented programming language. Slice expressions in this thesis are specified using the formal language LOTOS.

Jacobson [47,48] describes a method of object-oriented design for large real time systems. The method requires a designer to identify the blocks and components from which a system is constructed. The designer can then specify how the users interact with the system by means of a set of *use cases* which describe the event sequences required to implement the system functionality. Designers can begin with simple use cases identifying basic functionality, and then using a method of inheritance these basic use cases can be extended (or restricted) to define related use cases. Use cases which are related to each other

can also be combined in parallel. In this way, specifications can be built up in an incremental fashion until a complete behavioural specification is reached.

In Cockburn *et al* [17] protocols are specified by *message flow diagrams* to show how messages move through the protocol entities of the system. They formalize the notion of a message flow diagram and then use these diagrams, which represent incomplete behaviour of a system, to try and generate a set of prolog like rules which describe system behaviour. Their work focuses primarily on trying to generate a complete set of rules from an incomplete specification based on behaviour patterns involving sets of cooperating components.

Lavi and Kessler [59] propose using Statecharts to specify the behaviour of a set of concurrent components in a centralized behaviour description. Although they do not specifically advocate the use of slice expressions, they identify the problem of distributing such a centralized description among the concurrent components. They do not, however, propose a solution to this problem.

3.3 Chapter Summary

In summary, the main elements of a design environment being proposed are the following:

- Structural design which identifies components and interconnections.
- A structural design hierarchy where components can be recursively decomposed into subcomponents.
- Behavioural design which can be performed at any level of the structural design hierarchy.
- Behaviour specified either by means of slice expressions involving sets of components, or as component expressions which identify behavioural constraints on a single component.
- Specifying complex slice expressions by means of refinements to simpler slice expressions.

- Distributing the behavioural requirements of the slice expressions among the different components.

The remainder of this thesis will investigate the following issues:

- Representing the above design elements within a formal framework.
- Investigating how the formal representation can be used to assist in the design process.

Chapter 4: Design Representation

Chapter 2 discussed a number of issues related to representing structural and behavioural design; the presentation of that chapter was incomplete in a number of ways:

- The structural design representation was not formal.
- The behavioural design represented in LOTOS was not separable from the formal structural representation and was not fully developed for representing slice style specifications.

This chapter presents representations for concurrent system design, and notations for capturing the design. The major research contributions of this chapter are:

- Development of a structural design representation which is compatible with an event based behavioural design representation, and whose behaviour can be specified using either a resource oriented or a slice style (§4.1).
- Definition of a visual notation for representing structural design (§4.1.1.1).
- Integration of LOTOS with the structural design representation and then using LOTOS to express behaviour in both a resource oriented and a slice oriented style (§4.2.1.2).
- Definition of visual notations for representing behavioural design (§4.2.2). The visual notations are used to define and to refine slice expressions.

4.1 Representing Structural Design

This section introduces a formal representation of a system's structure, where structure is defined as the specification of the components from which a system is constructed, and the interconnection topology of the components. The primary motivation for developing this representation is to have a structural representation which is compatible with an event based behavioural representation and whose

behaviour can be specified using either a slice oriented or a resource oriented style of specifications.

A system's structure identifies the components from which a system is constructed and relations between these components. The primary relation of interest to this work is whether components communicate directly with each other during execution. If component **a** communicates with component **b** during execution then **a** is *connected* to **b**. A complete specification of the structural design of a system must therefore include:

- A description of the components of the system.
- A description of how the components of the system are interconnected.

Specifically not addressed by this research are a number of other relations which may also be defined as part of 'structure', for example: '**a** inherits from **b**' or '**a** is an instance of **b**'.

Each component has a *visible* part and a *hidden* part. The visible part of a component includes all the information a designer must know in order to connect the component to other components and understand the nature of the interaction; this includes: an *interface*; *functional behaviour*; and *temporal behaviour*.

- **Interface.** The interfaces to a component identify how a particular component is accessed, and how it accesses other components. An analogy can be drawn with hardware components. A hardware component has a set of lines and busses by which the component interacts with the outside world. Each specific line has a set of values which it can assume. Similarly, system components have a set of well defined interfaces allowing values to be passed between components. An interface to a component identifies the type of interaction in which the component is engaging, as well as the data values which are required during the interaction.
- **Functional behaviour.** The functional behaviour of a component identifies the data functions which are computed by the component. During the

execution of the system, data values will be received by a component from other components, new data values will be computed, and these new values will be passed on to other components. The receiving of data values from other components and the computing of new data values is the functional behaviour of a component.

- **Temporal behaviour.** In concurrent systems each component will require and impose various temporal constraints on the system. These constraints define when data is provided to a component and when and how components synchronize and communicate with each other. Such requirements define the temporal behaviour of a component.

The hidden part of a component includes all implementation dependent information which is not of concern to a designer wishing to use the component. This includes information such as internal structure, methods for representing data types, etc.

Structure must not only define the components and their interfaces but must also define how the components are interconnected. The components and their interfaces define what type of components exist and constraints on how they may be interconnected, e.g., a component which outputs an integer can only be connected to a component which inputs an integer. The *interconnection topology* defines how the components are “wired together”. The analogy in hardware is that a designer must not only specify which chips are being used to construct the system, but must also specify how the chips are connected to each other.

4.1.1 A Formal Representation of Structure

There are many different properties and characteristics which can be defined as being part of the structural design of a system [27,55]. In order to explore

the slice approach to concurrent system design, this research develops a structural representation with the following properties:¹:

- *Each component is active.* This research addresses the issue of complexity which is caused by temporal behaviour rather than complexity introduced by functional behaviour. Thus passive components, which are primarily used to represent data abstractions are not included within the structural representation.
- *Components encapsulate structure.* Any component can be decomposed into a set of interconnected subcomponents. Designing complex systems is often performed by hierarchically decomposing complex components. A structural design representation should therefore support structural decomposition.
- *Component interactions are synchronous.* Each interaction between components represents a synchronization between the components. If other interaction types are required (e.g., asynchronous[43,15], broadcast[34]), these can be implemented by introducing components into the design which support the appropriate interaction type.
- *A single interaction can involve any number of components.* Multiway interactions are a generalization of two way interactions and therefore provide more flexibility to designers. There is no direction to an interaction, as would be found for example in client/server models where the interaction is directed from the client to the server. During an interaction, data can pass between the components of the interaction in any direction and data functions can be computed and agreed upon by the components involved in the interaction.
- *Interaction points are not decomposable.* This restriction is imposed to make designs easier to analyze from a behavioural perspective.
- *Design structure is static.* Modeling of many systems requires dynamic structure where components are created and destroyed. Dynamic structure

¹ Some of the characteristics could also be defined as behavioural characteristics as well as structural characteristics. The decision to define them as structural is somewhat arbitrary.

is not included within the structural representation proposed here primarily to simplify the problem which is being addressed and to allow more time to explore the slice style of behaviour specification. Although this feature is not included it is a natural extension to provide in any future versions of the system.

The primary elements which identify structural design are the components and the points where the components interact. Both the components and the points of interaction are defined in terms of a more basic element called an *interaction point side* (IP side). An IP side defines constraints on a single component which is engaged in an interaction, essentially defining a component interface. It can be thought of as a ‘socket’ into which components can be ‘plugged’. A component can be plugged into an IP side only if it contains a plug compatible with the socket, i.e., only if it satisfies the constraints imposed by the IP side. An IP side specifies the data flow through the side during an interaction. Formally, an IP side is defined as follows.

Definition: Interaction Point Side.

An interaction point side is a tuple $\langle e, t \rangle$ where

- e is a unique identifier.
- t is a list of data types.

The identifier e provides a unique identification of the IP side.

Whenever an interaction occurs data flows between the components involved in the interaction. For an IP side $\langle e, t \rangle$, the set t represents the data types which must be provided by any component connected to that particular side. Any component which is plugged into side $\langle e, t \rangle$ must provide values of the correct type for each element of the list t whenever it engages in interaction at that side.

Components interact with each other at structural elements called *interaction*

points (IPs). Each interaction point represents a location where a multiway rendezvous can take place. An IP is made up of a list of IP sides where the number of sides represents the number of components which are involved in the multiway rendezvous. The sides are symmetric in the sense that components at one side do not “call” components at other sides; rather there is a mutual agreement among the components to engage in an interaction at the IP. Formally an IP is defined as follows.

Definition: Interaction Point (IP).

An *interaction point* (IP) is a pair $\langle e, e_{sides} \rangle$ where e is a unique identifier and e_{sides} is a list of interaction point sides.

The identifier e is used to provide a unique identifier for the interaction point.

The list of sides of the interaction point e_{sides} , defines the characteristics of an interaction which occurs at the IP. This includes:

- *Number of components.* An IP which consists of n sides represents an n -way rendezvous. The rendezvous will involve one component at each side of the interaction.
- *Dataflow.* The data types associated with an IP side identify data values which flow from that side to the other IP sides during an interaction. If for example a data type t is associated with side e_j of IP e , then when an interaction occurs at e the component at side e_j which is engaged in the interaction must provide an actual value of the correct type; any component at side e_k , $k \neq j$ can read the value provided during the interaction.

Components are the elements of a system which perform the ‘work’ and compute the data functions. From a structural perspective the only questions of interest when specifying components are:

- What are the interfaces of the component?

- How is the component interconnected to other components?

Both of these questions can be answered by identifying the IP sides to which a component is connected. Since an IP side identifies data flow, knowing the set of IP sides to which a component is connected completely defines the interface of the component. Each IP side is associated with exactly one IP; knowing the IPs which each component accesses completely defines the interconnection topology between components. Thus the only requirement for structurally specifying a component is to list the IP sides to which the component is connected. This leads to the following formal definition.

Definition: component

A *component* is a pair $\langle c, c_{sides} \rangle$ where:

- c is a unique identifier.
- c_{sides} is a set of IP side.

For a component $\langle c, c_{sides} \rangle$ the set c_{sides} identifies the IP sides to which a component is connected. Once the IP sides are known the complete interface to the component is defined and its interconnection to other components is specified. In particular a component c_1 can communicate with a component c_2 only if they are connected to different sides of the same IP; the data variables of the IP identify data flows which the components must support during the interaction.

A structural design specification must identify the components and the interconnection topology. This can be done by specifying the components and the IPs. Specifying the set of components identifies the number of components of a system (i.e., the number of active entities which can do work) and their interfaces. Specifying the set of IPs identifies the interconnection topology between the components. Thus a design structure can be formally defined as follows.

Definition: design structure.

A *structural design* is a pair $\langle IP, C \rangle$ where:

- IP is a set of interaction points
- C is a set of components

such that:

1. If $\langle e_1, [s_{1_1} \dots s_{1_n}] \rangle, \langle e_2, [s_{2_1} \dots s_{2_m}] \rangle \in IP$, $e_1 \neq e_2$ then the lists $[s_{1_1} \dots s_{1_n}]$ and $[s_{2_1} \dots s_{2_m}]$ do not contain any elements in common. (i.e., an IP side is an element of at most one IP.)
2. If $\langle c, S \rangle \in C$ and side s is an element of S then there exists an interaction point in IP containing s . (i.e., an IP side is an element of at least one IP.)

4.1.1.1 Visual Representation of Design Structure

One of the objectives of this thesis is to provide natural, intuitive and easy to use representation for design concepts. For design structures, it is quite natural to represent them graphically. This section provides a description of a visual notation which can be used to formally specify structural design.

For illustrative purposes, a simple producers/consumers system will be used as an example. The characteristics of the producers/consumers system are the following:

- The system's environment is modeled by two external component which are not part of the system being developed.
- Two producer components produce values of type ϵ . The producers are activated by an external stimulus *tick*.
- Two consumer components consume values of type ϵ . Upon consumption a consumer will notify an external component by means of *tock*.

- A buffer component provides buffering between the producers and the consumers.
- When the buffer is ready to accept a value it will accept a value from either producer.
- When the buffer is ready to provide a value to a consumer it will offer the value to whichever consumer is ready.

A structural design consisting of seven components and two IPs is shown in figure 4.1. The two producers are connected to the IP side *putin* which is used to provide a value to the buffer. Referring to the set I (the set of IP sides) it can be seen that the IP side *putin* has one data value of type t associated with it; this indicates that a producer must provide a data value of type t whenever it engages in an interaction involving IP side *putin*.

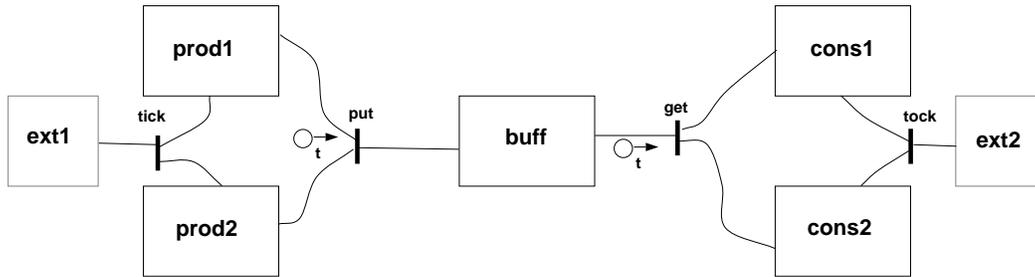
The buffer *buff* is connected to two IP sides: *putout* and *getin*. Looking at the set IP it can be seen that the sides *putin* and *putout* together form the IP *put*. The side *putout* does not have any data types associated with it and therefore the only data flow which occurs during an interaction at *put* is a transfer of type t from side *putin* to side *putout*.

The consumers and the buffer are connected in a similar manner using the IP *get*.

The visual notation uses icons to represent the various parts of the structural design. These icons are illustrated in figure 4.2.

Short thick lines represent IP sides; interaction point sides are then combined into interaction points. An n -sided event is represented by an n sided polygon. Data flow is included as part of the characteristics of each side. The IP identifier is written beside the IP.

Data flow at an interaction point side is represented by an arrow with an open circle at the tail. The data types associated with the flow are included as a list of labels on the arrow.



$I = \{ \langle tickin, [] \rangle,$
 $\quad \langle tickout, [] \rangle,$
 $\quad \langle tockin, [] \rangle,$
 $\quad \langle tockout, [] \rangle,$
 $\quad \langle putin, [t] \rangle,$
 $\quad \langle putout, [] \rangle,$
 $\quad \langle getin, [t] \rangle,$
 $\quad \langle getout, [] \rangle \}$

$IP = \{ \langle tick, [tickin, tickout] \rangle,$
 $\quad \langle tock, [tockin, tockout] \rangle,$
 $\quad \langle put, [putin, putout] \rangle,$
 $\quad \langle get, [getin, getout] \rangle \}$

$C = \{ \langle prod1, \{putin, tickin\} \rangle,$
 $\quad \langle prod2, \{putin, tickin\} \rangle,$
 $\quad \langle buff, \{putout, getin\} \rangle,$
 $\quad \langle cons1, \{getout, tockout\} \rangle,$
 $\quad \langle cons2, \{getout, tockout\} \rangle$
 $\quad \langle ext1, \{tickout\} \rangle,$
 $\quad \langle ext2, \{tockin\} \rangle \}$

Figure 4.1 Producers/consumers problem – structural design specification.

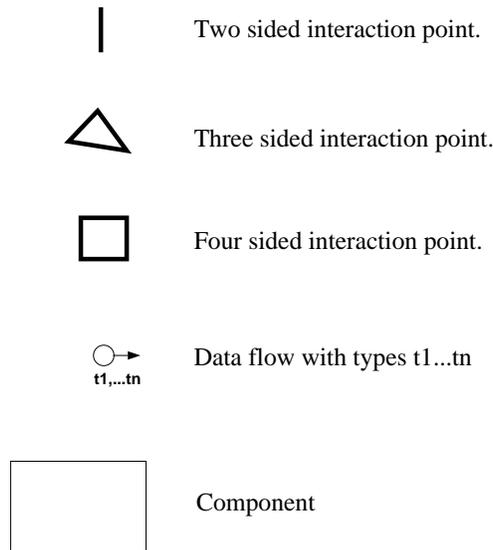


Figure 4.2 Graphical icons for structural design.

A component is represented as a rectangle. The identifier of the component is included as a label inside the rectangle. Components are constructed from sets of IP sides. If IP side s is connected to component c this is indicated by drawing a line from component c to s .

In the graphical representation of figure 4.1, the two producers are represented by the components `prod1` and `prod2` and the two consumers by `cons1` and `cons2`. The buffer between them is represented by the component `buff`. Four IPs are used: `tick` and `tock` to interface to the external components; `put` to pass data from the producers to the buffer; and `get` to pass data from the buffer to the consumers. The data flow indicates a single data value of type τ being provided by the producers for the buffer at IP `put`; and by the buffer for the consumers at IP `get`.

4.1.1.2 Decomposition of Components

Structural design of concurrent systems can often be simplified by organizing the structure in a hierarchical manner, first subdividing a system into large compo-

nents and then subdividing these large components into finer grained components. This process of subdividing components into finer grained components continues until the primitive components which can be implemented as single artifacts are found. Since this process of hierarchical decomposition is integral to the approach taken by many design methodologies, any formal representation of structure must be able to accommodate a hierarchical design.

In order to represent the hierarchical design process a system is represented not by a single structural design $\langle IP, C \rangle$, but rather by a set of structural designs $\{\langle IP_1, C_1 \rangle, \dots, \langle IP_n, C_n \rangle\}$ where each element of the set represents the design at a particular level of the design hierarchy. For example, assume that the structure of a system S is to be specified. The first step is to create the top level structural specification $\langle IP_S, C_S \rangle$ which defines the structural decomposition of S . Given that $C_S = \{C_{S1}, \dots, C_{Sn}\}$ structural specifications can be created for C_{S1}, \dots, C_{Sn} which give the decomposition of the components of S . The decomposition process then proceeds to the required level of granularity.

A system's structural specification requires a means of organizing these structural designs into a design hierarchy, i.e., which structural design is the top level design, and for a particular component c which structural design $\langle IP_c, C_c \rangle$ represents the decomposition of c . The hierarchy of structural designs can be represented as a tree (figure 4.3). Each node of the tree is labeled with a component identifier and a structural design: the component identifier indicates which component is being specified at the node (with the overall system being considered a single component); the structural specification gives the internal structure of the component.

Formally, a structure hierarchy tree can be defined as follows.

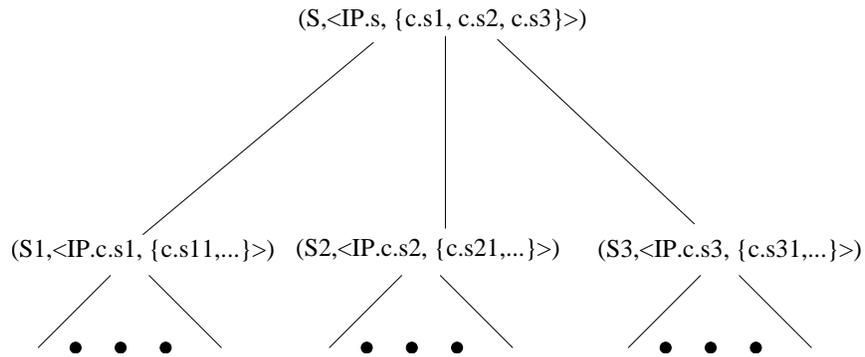


Figure 4.3 Example of a hierarchical decomposition of components.

Definition: Structure Decomposition Tree

A *structure decomposition tree* is a tree with the following properties:

- Each node of the tree has a label of the form (C,S) where C is a component identifier and S is either a structural design or the identifier *primitive*;
- A node of the tree is labeled $(C,primitive)$ if and only if it is a leaf node.
- If x is a node labeled (C_1,S_1) and y is a node labeled (C_2,S_2) then $C_1=C_2$ if and only if $x=y$.
- If x is a node labeled (C,S) where S is a structural design containing components $C_1...C_n$ then x has exactly n children and the labels of these children are $(C_1,S_1)...(C_n,S_n)$.

At the root of the tree is the structural design for the system. For a node which gives the structural decomposition of some component c , any children of the node represent structural decompositions of some subcomponent of c .

An example structural decomposition is given in figures 4.4 and 4.5. Figure 4.4 is the producer/consumer problem with the `buff` and the consumer components decomposed into two components each. The resulting structure tree is shown in figure 4.5.

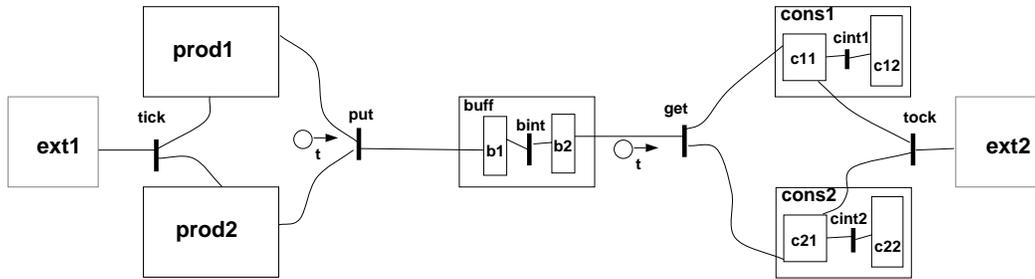


Figure 4.4 Producers/consumers problem – Structural decomposition.

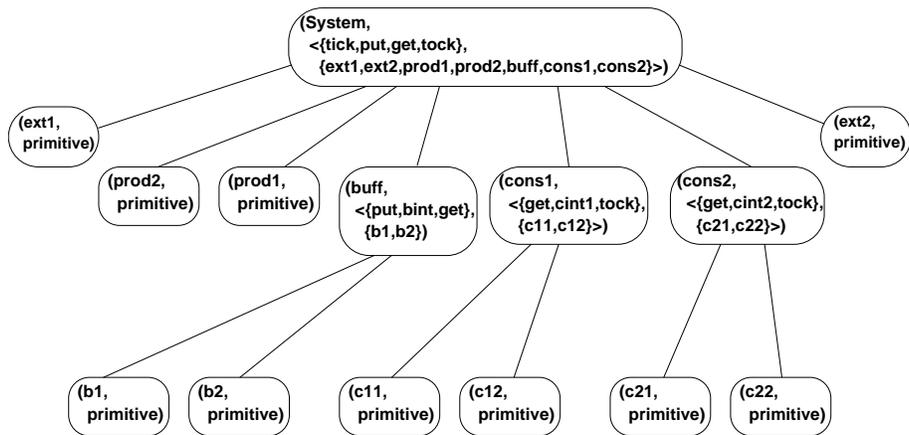


Figure 4.5 Producers/consumers problem – Structural decomposition tree.

4.1.2 Discussion

This section developed a representation for structural design and defined a visual notation which can be used to capture the structural design. There are numerous examples of other research in design representation which model design structure. This section looks at a number of structural representations which are currently being used for concurrent system design.

MachineCharts [10,12] is a graphical based method for the design of real time systems. Components are defined as *boxes* where a box can be either active (such as Ada tasks) or passive (such as abstract data types). Component interactions are either extended rendezvous (in the case of interactions between active components) or procedure calls in the case of an interaction between an active and a passive component. Interactions are always two-way, although a multiway interaction can be simulated using the extended rendezvous interaction. Hierarchical decomposition is achieved by defining a structural box and then decomposing the box into interconnected boxes. Dynamic components are also supported. A visual notation provides a means of identifying the boxes and interconnections of the system. The visual notation is much richer than the one presented in this section, in that it includes features such as timeouts, internal machinery of primitive components, etc.

The TELOS project [78,77] defines structure in terms of a set of *actors* and *ports* through which actors exchange messages. Each actor is similar to a component in that it represents an independent, active, component. Actors can be hierarchically decomposed into interconnected sets of actors. The structural design of TELOS is richer than the design presented here in a number of ways. For example, TELOS has the following features:

- *Dynamic structure.* TELOS allows (in a restricted way) the dynamic creation and deletion of actors.

- *Intercomponent communication.* TELOS does not specify a priori the characteristics of the intercomponent communication. Interactions between components can be synchronous, asynchronous, or involving a more complex protocol. A single interaction is however restricted to two components.
- *Hierarchical decomposition.* TELOS allows for hierarchical decomposition of actors. However, the decomposition is not represented by a tree, but rather by a lattice. This allows a single actor to be a subcomponent of more than one actor.

TELOS provides a visual notation for representing the design structure. The notation is similar to the notation of this research with rectangles representing actors and lines representing connections between actors. Within TELOS, every interconnection between actors is represented by a fixed binding between ports; there is no concept comparable to an interaction point which can be used to simplify interconnections between many components.

The Raddle/Verdi project [22,21,29,28] is a specification language (Raddle) together with a visual notation (Verdi). The components can be either active or passive. Interactions between components are synchronous, and multiway interactions are supported. Hierarchical decomposition of structure is also supported.

4.2 Representing Behavioural Design

A design of a system requires not only a structural specification but also a behavioural specification. This section:

- Defines what is meant by the *behaviour* of a design structure.
- Shows how LOTOS can be used to represent the behaviour of a design structure, either in a resource oriented or a slice oriented style.
- Develops visual notations of behaviour which are useful for defining and refining slice style specifications.

4.2.1 Behaviour of a Design Structure

Section 4.1 developed a formal representation for specifying design structure. We must now turn to the question as to what is meant by the ‘behaviour’ of a particular design structure. When a designer specifies the behaviour of a design what specifically is being defined? This section will answer these questions. Section 4.2.1.1 will define more precisely what is meant by the behaviour exhibited by a structure; section 4.2.1.2 will then show how LOTOS can be used as the language to define the structures behaviour.

4.2.1.1 Integrating Behaviour and Structure

Given a design structure what does it mean when we talk about the “behaviour” of this structure? What are the aspects of behaviour which are significant?

One can view a designer as having a set of “probes” inserted at each interaction point of a structure. Each probe allows a designer to observe what is happening at the corresponding IP. By characterizing all possible observations which an observer may see by watching these IP probes, a designer can specify the behaviour of a concurrent system.

What is it that an observer can see at each IP probe? Clearly one of the things that must be observable is the occurrence of an interaction between components of the system. An observer watching the probes can detect every interaction which occurs in the system. By recording the history of all the interactions which occur between the components, an observer can record a *trace* of the system execution. One method of specifying the system is to characterize all possible valid traces of system behaviour. For example figure 4.6 is a design of a simple buffer which inputs at `get` and then outputs either at `put1` or `put2`. The behaviour of this buffer can be represented by defining the set of valid traces; a partial list of the traces is given in figure 4.7.



Figure 4.6 Design structure example — a buffer.

Trace of interactions
λ
<code>get</code>
<code>get.put1</code>
<code>get.put2</code>
<code>get.put1.get</code>
<code>get.put2.get</code>
...

Figure 4.7 Observations at IPs showing possible traces.

However, if the only thing which can be observed by a probe is whether an interaction occurs or not, there are important properties of a component which cannot be determined simply by observing the probe. The specification of the system `buff` which is based on traces misses an important characteristic of the system, namely who is it that makes the decision as to whether a `put1` interaction or a `put2` interaction will follow the `get`? Two options are possible:

- After engaging in `get` the component `buff` makes a decision as to which of the `put` interactions will be used; `ext2` will accept whichever `put` event is offered by `buff`. How `buff` makes the actual decision as to which `put` event will be

offered can be left unspecified at this stage of the design process since we do not wish to specify the actual algorithm which will be used.

- After engaging in `get` the component `buff` must be ready to engage in both `put1` and `put2`; the decision is left up to `ext2` to determine which of the two `put` interactions will be used.

Whichever one of the two options is chosen will have a significant impact on the implementation of `buff`. However simply by observing interactions occurring at the IP probes an observer cannot determine which is the intended behaviour of `buff`.

In order to be able to observe a difference between the two possible implementations of `buff` it is necessary to extend the power of each IP probe: not only can a probe observe when an interaction occurs at an IP, but it can also observe whether a component connected to the IP is ready to engage in an interaction at the IP. Thus an observer can see whether a component is *offering* a particular interaction at an IP or *refusing* (i.e., not offering) the interaction at the IP.

Figure 4.8 shows in tabular form some of the observations an observer may see by watching the IPs of the component `buff` assuming that both occurrences of interactions as well as *refusals* are visible at each probe (the rows of the table are not sequential in time). The left column shows the traces which have been observed; the right column are the refusals which may be seen after the trace has been observed. The assumption made in these observations is that it is the `buff` and not `ext2` which makes the decision as to which `put` event will be used after a `get`. Note that for some traces (e.g., `get`) there are two entries in the table representing two possible sets of refusals which may be observed. Which actual observation is made is left as a nondeterministic choice of the component `buff`.

Formalizing Behaviour and Structure. The basic structural elements from which a structural design is constructed are components and interaction points. In order to define behaviour it is necessary to define a basic behavioural element.

Trace of interactions	Interactions refused by <code>buff</code>
λ	{ <code>put1</code> , <code>put2</code> }
<code>get</code>	{ <code>get</code> , <code>put1</code> }
<code>get</code>	{ <code>get</code> , <code>put2</code> }
<code>get.put1</code>	{ <code>put1</code> , <code>put2</code> }
<code>get.put2</code>	{ <code>put1</code> , <code>put2</code> }
<code>get.put1.get</code>	{ <code>get</code> , <code>put1</code> }
<code>get.put1.get</code>	{ <code>get</code> , <code>put2</code> }
<code>get.put2.get</code>	{ <code>get</code> , <code>put1</code> }
<code>get.put2.get</code>	{ <code>get</code> , <code>put2</code> }
...	

Figure 4.8 Observations at IPs showing possible refusals after a trace.

The basic behavioural element is an *interaction* where an interaction is a single n -way rendezvous occurring at an n sided IP. The information which must be represented as part of the interaction is:

- The interaction point where the interaction occurred.
- The components involved in the interaction.
- The data values transferred between components during the interaction.

Formally, an interaction can be defined as follows.

Definition: Interaction.

An *interaction* is a tuple $\langle i, c, V \rangle$ where:

- $i \in IP$ is the interaction point where the interaction occurred.
- c is a list of components such that:
 - The number of elements in c equals the number of IP sides in i .
 - For all k such that $1 \leq k \leq length(c)$ the k th component of c is connected to the k th side of IP i .
- V is a list of data value lists and for all k such that $1 \leq k \leq length(c)$ the values of the k th list of V correspond to the data types of the k th side of IP i .

For each interaction $\langle i, c, V \rangle$, i represents the IP where the synchronization occurred. The list c represents the list of components involved in the interaction. The particular component involved at a particular side is determined by the ordering of components in the list c and the ordering of sides of the IP i , i.e., the k th component of c was the one involved at the k th side of i .

The lists V represent the data values passed between the components. The ordering of the list determines the values which were provided at each side of the IP i , i.e., the k th data value list of V defines the data values which were contributed by the component at the k th side of i .

Figure 4.9 illustrates two interactions involving the producer/consumer structure of figure 4.1. The first interaction represents *prod1* transferring value 5 to *buff* at IP *put*. The second interaction represents *buff* transferring value 5 to *cons1* at IP *get*.

4.2.1.2 Using LOTOS as a Behavioural Design Language

The basic unit of behaviour in LOTOS is the *event* where an event consists of a gate name and a list of typed data values. The LOTOS event can be written

$$\langle \text{put}, [\text{prod1}, \text{buff}], [[5], []] \rangle$$

$$\langle \text{get}, [\text{buff}, \text{cons1}], [[5], []] \rangle$$

Figure 4.9 Example interactions for the producer/consumer structure.

as $g!v_1!v_2\dots!v_n$ where g is the gate name and the v_k are the data values.

The basic unit of behaviour for a structural design is an *interaction* $\langle i, c, V \rangle$, which consists of an IP identifier i , a list of component names c , and a list of lists of typed data values V exchanged during the interaction.

In order to use LOTOS as the behavioural design language it is necessary to be able to represent component interactions as LOTOS events. A straightforward mapping between an interaction $\langle i, c, V \rangle$ and a LOTOS event e can be done as follows:

- The interaction point identifier i is the LOTOS gate name.
- The event e has $|c| + \text{number}(V)$ data values associated with it where $\text{number}(V)$ is the total number of data values in the lists of V .
- The first $|c|$ data values associated with event e are the identifiers of the components which synchronize at IP i ; the position of the identifier in the data value list can be used to associate a component with a particular side of the IP.
- The last $\text{number}(V)$ data variables associated with gate e correspond to the data values which are exchanged between components when they synchronize at IP e .

Using the above representation, the interaction $\langle \text{put}, [\text{prod1}, \text{buff}], [[5], []] \rangle$ of the producer/consumer structure of figure 4.1 is represented as the LOTOS event:

```
put !prod1 !buff !5.
```

Using a Resource Oriented Style in LOTOS. In a resource oriented style of specification a designer organizes a behavioural specification as follows:

- For each component $c_1, c_2 \dots c_n$ of the system generate a LOTOS expression $P_{c_1}, P_{c_2}, \dots P_{c_n}$ which represents the behaviour of the corresponding component.
- Using the algebraic operators of LOTOS combine the expressions $P_{c_1}, P_{c_2}, \dots P_{c_n}$ to represent the overall behaviour of the system. The algebraic operators used to combine the expressions are chosen to represent the interconnection topology of the components.

An example of a resource oriented specification for the producer/consumer structure is shown in the partial LOTOS specification of figure 4.10. The process `producer` defined at lines 13 through 17 describes the behaviour of a generic producer component. The `producer` process requires two parameters: `name` which is the component identifier of the producer component; and `dest` which is the identifier of the component to which the producer sends its data. The process `consumer` (lines 18 to 22) can be defined in a similar way. The process `buffer` (lines 23 to 32) contains the single parameter `name` which is the identifier of the buffer component. The process invocations at lines 3, 5, 7, 9, and 11 have the component names passed as part of the invocation. Note that each event as it occurs will contain the identifiers of the components which are involved in the corresponding interaction. (This specification is similar to the LOTOS specification in figure 2.11 of Chapter 2. The main difference is that the specification of figure 4.10 includes the component names as part of the events.)

The Wiring Problem.

One problem with LOTOS which was encountered when using the resource oriented style of specification was the inability to represent the behaviour of any arbitrary network of components where the method of component interaction is synchronous rendezvous. The problem is illustrated by the structure of figure

```

1  behaviour
2  hide put,get in
   {*start up two producers called proc1 and proc2 and two consumers
   called cons1 and cons2 - no synchronization*}
3  (producer[tick,put] (proc1,buff)
4  |||
5  producer[tick,put] (proc2,buff)
6  |||
7  consumer[get,tock] (cons1,buff)
8  |||
9  consumer[get,tock] (cons2,buff)

   {*Buffer synchronizes with producers and consumers on put and get*}
10 | [put,get] |
11  buffer[put,get] (buff)

12 where
13 process producer[tick,put] (Name, Dest:Component) :noexit :=
14   tick ?C1:Component !Name;
15   put !Name !Dest ?N:Nat;
16   producer[tick,put]
17 endproc

18 process consumer[get,tock] (Name, Src:Component) :noexit :=
19   get !buff !Name ?N:Nat;
20   tock !Name ?C:Component;
21   consumer[get,tock]
22 endproc

23 process buffer[put,get] (Name:Component) :noexit :=
24   buff[put,get] (Name)
25   |||
26   buff[put,get] (Name)
27   where
28     process buff[put,get] (Name:Component) :noexit :=
29       put ?C1:Component !Name ?N:Nat;
30       get !Name ?C2:Component !N;
31       buff[put,get] (Name)
32     endproc
33 endproc

```

Figure 4.10 Example of resource oriented style in LOTOS.

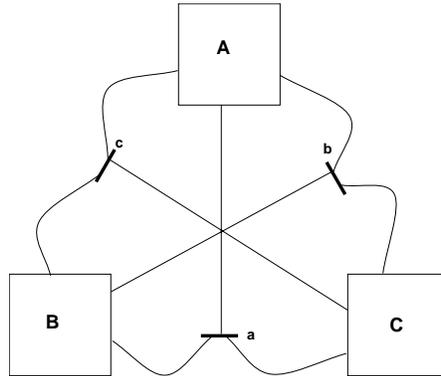


Figure 4.11 Structure of system whose behaviour cannot be directly represented in a resource oriented style.

4.11. The system is constructed from three components A , B , and C , and three interaction points a, b, c . A and B synchronize on a and b and execute in parallel on c ; A and C synchronize on a and c and execute in parallel on B ; and B and C synchronize on b and c and execute in parallel on a . Using a resource oriented style of specification a designer can construct processes P_A , P_B , P_C to represent the behaviour of the components A, B, C . However, since the LOTOS synchronization operator is not associative, these processes cannot be combined using the synchronization operator in a manner which reflects the overall system behaviour. Since A and B synchronize on a and b , their combined behaviour is given by $P_{AB} = (P_A \parallel [a, b] P_B)$. When P_C is composed with P_{AB} to represent the system behaviour, neither the expression $(P_C \parallel [a, b, c] P_{AB})$ nor the expression $(P_C \parallel [c] P_{AB})$ correctly represent the intended behaviour; the former implies that a and b require three way synchronizations between A , B , and C ; the later implies that C does not synchronize with A on a and with B on b .

Solutions to representing arbitrary component networks in LOTOS usually involve constraining in some way the system structures which can be represented, such as by requiring asynchronous communication between components (see for example [32]). In order to maintain generality in the design process however, it is

desirable to retain the use of rendezvous interactions between components. This requires a technique of composing component specifications which overcomes the wiring problem. The problem can be stated as follows.

- Given
 - a design structure $S = \langle IP, C \rangle$.
 - for each $c \in C$ a LOTOS behavior expression B_c which represents the behaviour of component c .
- Construct a LOTOS behaviour expression B_S which represents the resource oriented behaviour of S .

The method proposed by this research for constructing B_S is to first define a LOTOS process `default[E] (c:Component)` where **E** is a list of all the IP identifiers and `Component` is a data sort which includes all the component identifiers of the system. The behaviour of `default` can be described as follows:

- `default[E] (c)` will continuously offer event e if and only if e represents an interaction in which component c is not involved.

Thus `default[E] (c)` will engage in any event which represents an interaction in which component c is not involved and will refuse to engage in any event representing an interaction involving component c .

Given that $C = \{c1, c2, \dots, cn\}$, the desired behaviour expression B_S can now be constructed as:

$$\begin{aligned}
 B_S = & \\
 & (B_{c1} \parallel \text{default[E]}(c1)) \parallel \\
 & (B_{c2} \parallel \text{default[E]}(c2)) \parallel \\
 & \dots \\
 & (B_{cn} \parallel \text{default[E]}(cn))
 \end{aligned}$$

To understand the behaviour of B_S , note that every line of the expression synchronizes on every event. If component c_j is involved in the interaction represented

by the event then the expression B_{c_j} (or one derived from it) will synchronize with the event; else the expression `default[E](c_j)` will synchronize.

Construction of the expression for B_S depends on the construction of the process `default[E](C:Component)`. Given a structural design $\langle IP, C \rangle$, the process `default` can be generated from the set of interaction points I . The method for doing this is as follows:

- For each $\langle e, e_{sides}, e_{vars} \rangle \in IP$ define a process `default_e[e](C:Component)` as follows:

```

process default_e(C:Component) : noexit :=
  e ?c1 ?c2 ... ?cn ?d1 ?d2 ...?dm
  [(c1 ne C) and
   (c2 ne C) and
   ...
   (cn ne C)];
  default_e(C)
endproc

```

where $|e_{sides}|=n$ and $|e_{vars}|=m$.

- The process `default` can now be defined as follows:

```

process default[e1,...,ek](C:Component) : noexit :=
  default_e1[e1](C) |||
  default_e2[e2](C) |||
  ...
  default_ek[ek](C)
endproc

```

where $e_1...e_k$ are the interaction point identifiers.

Using the above method, a behaviour expression representing the behaviour of the structure of figure 4.11 can be constructed as shown in figure 4.12. Note

```

(PA ||| default[a,b,c] (A)) ||
(PB ||| default[a,b,c] (B)) ||
(PC ||| default[a,b,c] (C))
where
  process default[a,b,c] (X:Component) : noexit :=
    default_e[a] (X) |||
    default_e[b] (X) |||
    default_e[c] (X)
  endproc
  process default_e[e] (X:Component) : noexit :=
    e ?X1 ?X2 [(X1 ne X) and (X2 ne X)];
    default_e[e] (X)
  endproc
  .
  .
  .

```

Figure 4.12 LOTOS resource oriented specification for structure of figure 4.11

that once a structural specification has been created and the component behaviours defined, the generation of the `default` process and the composition of the processes into a behaviour expression representing the system behaviour can be automated. Note also that the above technique for overcoming the wiring problem requires that interactions be mapped to LOTOS events as described in §4.2.1.1, with the component names involved in each interaction being encoded as data values of the LOTOS event.

Using a Slice Style in LOTOS. The slice style for specifying behaviour of interconnected components identifies behaviour patterns which involve sets of components. Such a style can be accommodated in a straightforward manner using the constructs of LOTOS. One approach is to represent each slice expression as a LOTOS behaviour expression. More complex slice behaviours can be represented by combining these simpler slice expressions into more complex behaviour patterns by using the LOTOS operators.

```

specification slice1[tick,tock,put,get]
behaviour
  pcslice[tick,tock,put,get](1,prod1,cons2)

where
  process pcslice[tick,tock,put,get]
    (N:nat, P,C:component) :exit :=
    tick !ext1 !P;
    put !P !buff !N;
    get !buff !C !N;
    tock !C !ext2;
    exit
  endproc

```

Figure 4.13 Example of a slice style for a the producer/consumer example.

As a simple example, assume we wish to define a slice expression for the producer/consumer structure (figure 4.1, page 75) which illustrates the following interaction sequence as a possible example of system behaviour:

- A *tick* interaction is accepted by `prod1`.
- `prod1` passes the value 1 to `buff`.
- `buff` passes the value 1 to `cons2`.
- `cons2` outputs a *tock* interaction.

A LOTOS expression representing this behaviour is shown in figure 4.13 (a visual representation of the process `pcslice` is shown in figure 4.14). The process `pcslice` sequences the four interactions. The behaviour expression `slice1` is an instantiation of this process with the appropriate values. Note that the expression `slice1` does not represent the behaviour of a single component of the system, but rather defines a behaviour pattern which crosses component boundaries.

Having defined the simple slice `slice1`, a designer may wish to represent the behaviour of two slices executing concurrently, for example with one slice passing a value 1 from `prod1` to `cons2` and another slice passing value 2 from `prod2` to

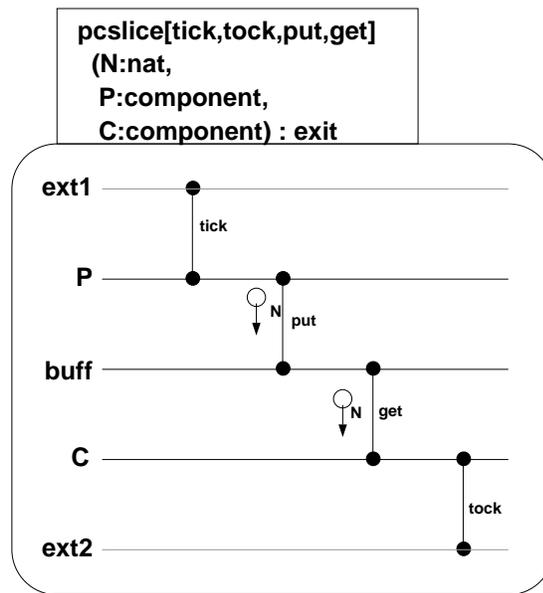
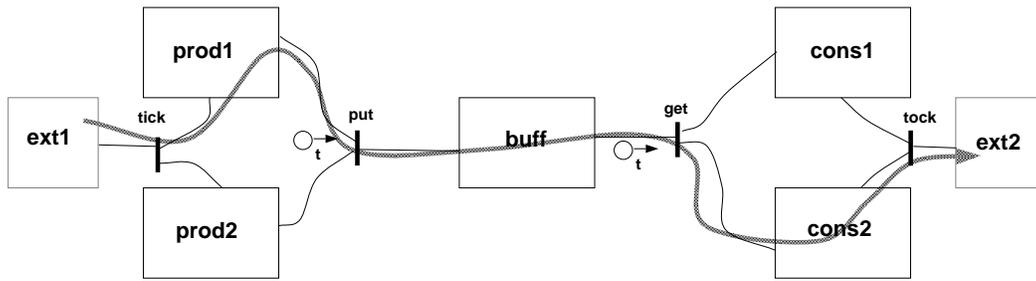


Figure 4.14 Producer/consumer slice example in a visual notation.

`cons1`. This can be done by instantiating two copies of the process `pcslice` and combining them using the LOTOS interleaving operator `|||` as follows:

```
specification slice2[tick,tock,put,get]
behaviour
pcslice[tick,tock,put,get](1,prod1,cons2)
|||
pcslice[tick,tock,put,get](2,prod2,cons1)
```

4.2.2 Visual Notations

In order to provide a more natural and intuitive representation of behaviour, a number of visual notations will be defined in this section for use in this research. The purpose of these visual notations is not to create a complete graphical syntax for LOTOS (as is done for example in [45]). Rather it is to develop visual notations which can be used at specific points and for specific purposes in the design process. Thus the visual notations for behaviour are not intended as general purpose behaviour specification methods; their objective is to emphasize and highlight a particular aspect of behaviour in a way which is easy to understand and to make the methods more accessible to engineers without a significant training in formal methods such as LOTOS.

Two graphical notations will be introduced here: *timelines* and *state transition diagrams*. Timelines are used to define simple slice expressions. A notation for combining timelines is used to facilitate behaviour refinement by slice extension as described in §3.1.2.

State transition diagrams are a visual notation for generating a state oriented LOTOS specification. They are used in this research primarily to specify synchronization processes when refining slices by combination as described in §3.1.2.

4.2.2.1 Timeline Representation of LOTOS Processes

Timelines are a graphical notation for representing the behaviour of sets of interconnected components. A basic timeline is quite simple and intuitive; an example was shown in figure 4.14. Visually a timeline is represented as a set of parallel horizontal lines called *axes* with time increasing in the horizontal direction. Each axis represents the behaviour of a single component of the system. An annotation at the left end of the axis indicates which component's behaviour is being represented. Axes drawn as dotted lines represent external components which are not part of the system under development.

Interactions are represented as thick vertical lines; a filled circle is placed at the intersection of the interaction line and the axis of a component involved in the interaction. A label beside the interaction line gives the identifier of the IP where the interaction occurred.

Data flow is indicated by an annotation similar to that used in structural diagrams. Data flow which occurs during an interaction is represented by a short arrow flowing out of the component axis which provides the data value. An annotation on the timeline indicates the value of the data.

Timelines as just described are rather limited in their expressiveness using only simple sequences of interactions. In order to increase the expressiveness three additional features are added to the basic timeline:

- **Parameterization.** A designer can define a timeline with a set of typed parameters which can be passed to the timeline. A parameter can represent either a data value passed between components, or a component name (i.e., a parameter of type `Component`). Within the timeline the formal parameters are treated as constants and can be used to label component axes or to label dataflow arrows.
- **Local Variables.** As part of the timeline the designer can specify a set of typed local variables. Each variable represents a value which will be computed

during the execution of the timeline. Once a variable has been declared it can be used as a label on a component axis or as a data value passed between components.

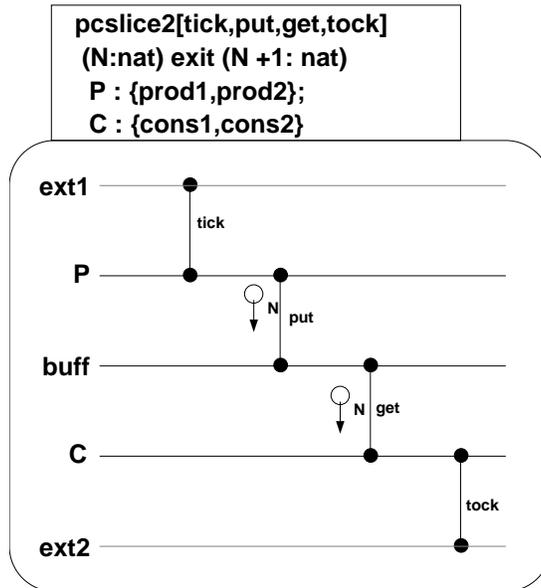
A variable becomes instantiated when it is first used in a timeline; this can either be when a component axis labeled with a variable is involved in an interaction or the first time the variable appears as part of a dataflow.

Every variable can have an optional boolean condition associated with it which can be used to restrict the set of values which the variable can assume: valid values for the variable are those which cause the condition to evaluate to *true*. The boolean condition is an expression built up from constants, parameters, and variables which are instantiated before the variable associated with the condition.

- **Exit Values.** Once a timeline completes execution it exits with a list of typed values. These values can be expressions involving constants, parameters, or variables. The exit values of a timeline are particularly useful when combining timelines together where the exit values of one timeline become the input parameters of the next timeline.

Figure 4.15 illustrates a timeline which includes parameters, variables, and exit values. The timeline represents the behaviour of the producer/consumer example and includes the following features:

- A parameter `n:nat` is passed to the timeline. This parameter will represent the value produced by the producer.
- The variable `p` is a placeholder for the name of a producer; it can assume the value `prod1` or `prod2` and captures the feature that either producer can engage in the *tick* interaction and pass the value on to the buffer. The variable `c` is the corresponding placeholder for the consumer component name.
- An exit value of type `nat` is output. The value output is equal to `n+1`.



```

process pcslice2[tick,put,get,tock] (N:nat) : exit(nat) :=
  tick !ext1 ?P:component[P in {prod1,prod2}];
  put !P !buff !N;
  get !buf ?C:component !N [C in {cons1,cons2}];
  tock !C !ext2;
  exit(N+1)
endproc

```

Figure 4.15 Timeline example for producers/consumers structure including parameters and exit values.

A timeline has a very natural representation as a LOTOS process. The LOTOS process corresponding to the timeline of figure 4.15 is shown below the timeline.

Constructing Timelines from Segments. Timelines are a graphical method for representing slice expressions. In order to enhance the expressiveness of timelines and facilitate the process of refining slices by extension, a set of operators is defined which allows a designer to express more complex behaviour patterns from simpler timelines by constructing timeline segments into more complex slice expressions. These operators are based on the LOTOS operators. Graphical notations will be provided for those operators which are used in examples later in this thesis. The operators used are:

- **choice.** The choice operator allows a designer to specify that the system makes a choice between a number of different timelines.
- **enable.** The enable operator permits a designer to string timelines together to form longer behaviour patterns. If *timeline1* enables *timeline2* then once *timeline1* has successfully completed *timeline2* begins executing.
- **disable.** If *timeline1* disables *timeline2* then while *timeline2* is executing and has not successfully completed it can be interrupted by *timeline1*; at that point *timeline2* stops executing and *timeline1* will start executing.

Visual notations are illustrated by example in figure 4.16. The corresponding LOTOS definitions are included to the right of each operator.

Figure 4.17 defines a process *twoslices*. It is constructed from the process *pcslice2* of figure 4.15. This process makes a choice between executing *pcslice2* with value 1 followed by *pcslice2* with value 2; or executing *pcslice2* with value 0 followed by *pcslice2* with value 1. The corresponding LOTOS process definition is given below the diagram.

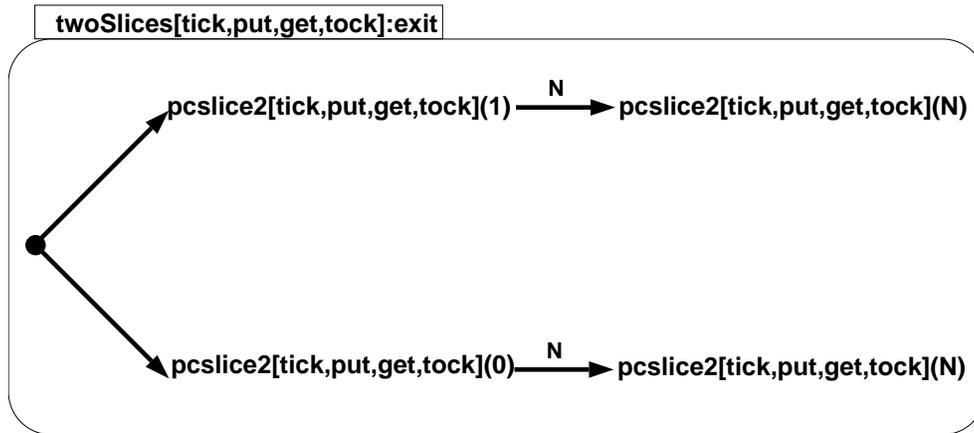
name	graphical icon	LOTOS representation
enable		$tl1[\dots](p1,p2) \gg$ $\text{accept } q1:\text{type1 } q2:\text{type2 in } tl2(q1,q2)$
disable		$tl1[\dots](p1,p2) [\>$ $tl2(q1,q2)$
choice		$tl1[\dots](p1,p2) []$ $tl2[\dots](q1,q2) []$ \dots $tlx[\dots](r1,r2)$

Figure 4.16 Graphical icons for timeline operators.

4.2.2.2 Representing State Transition Diagrams as LOTOS Processes.

State transition diagrams are a well known graphical technique for representing behaviour. This research uses state transition diagrams to represent synchronizing processes, which are constraints on slice expressions combined in parallel. As shown by Pnueli in [74] a large class of constraints can be expressed by means of an automata. By using a state oriented style in LOTOS, these constraints can be represented as state transition diagrams; LOTOS operators can then be used to form the conjunction of the constraints.

In order to specify a state machine, a visual notation for representing state transition diagrams was developed as part of this research; a transformation from the visual state transition diagram representation to a LOTOS process definition was also defined. This section summarizes some features of the visual notation



```

process twoSlices[tick,put,get,tock] : exit :=
  (pcslice2[tick,put,get,tock](1) >>
    accept N:nat in pcslice2[tick,put,get,tock](N)
  []
  (pcslice2[tick,put,get,tock](0) >>
    accept N:nat in pcslice2[tick,put,get,tock](N)
  )
endproc

```

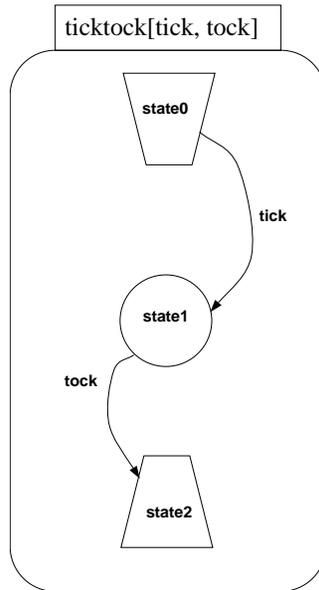
Figure 4.17 An example of combining timelines.

and the transformation to LOTOS. More complete details can be found in [57]. In conjunction with this research a tool to draw state transition diagrams and automatically transform them to LOTOS was developed [57].

A state machine consists of the following elements:

- A set of states S .
- An initial state s_0 in S .
- A labeled set of transitions T which is a subset of $S \times S$.
- A set of exit states S_e which is a subset of S .

An example of a graphical representation of a state machine is shown in figure 4.18 which represents an extensional view of the producer/consumer structure, with a



```

process ticktock[tick, tock] : exit :=
    tick;
    tock;
    exit
endproc

```

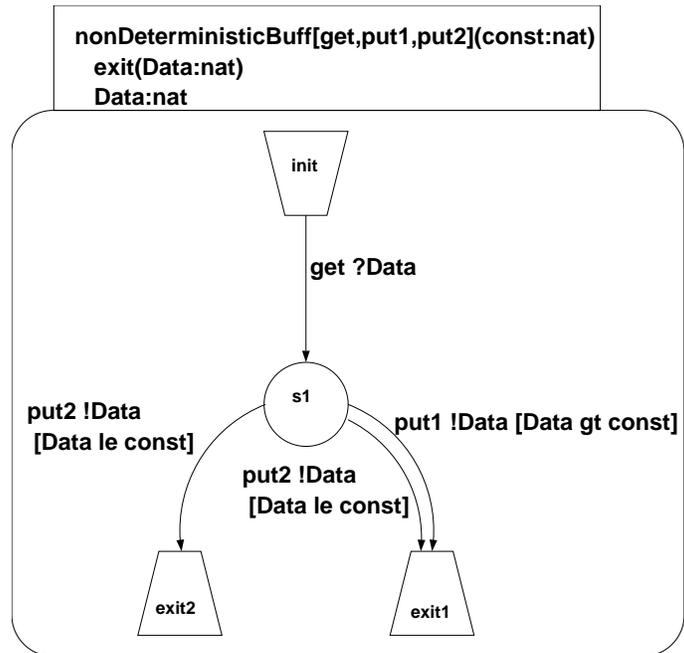
Figure 4.18 Example of a state transition diagram.

single *tick* followed by a single *tock*, followed by an exit from the state machine. The elements of the state machine are represented graphically as follows:

- The initial state `state0` is represented as a trapezoid with the shortest side on bottom.
- The exit state `state2` is represented as a trapezoid with the shortest side on the top.
- States which are not initial nor exit states (`state1`) are represented as a circle.
- Transitions are represented as arcs joining states. The arc is annotated with the event causing the transition.

The basic state machine is rather limited in its expressive power. In order to extend its expressiveness a number of features are added to basic state machines. These features are described below and illustrated in figure 4.19. The state transition diagram of 4.19 defines a process called `nonDeterministicBuff` and is associated with the buffer structure of figure 4.6 (page 82). The state machine interacts on three gates (`get`, `put1` and `put2`) and is passed one parameter when invoked (the parameter `const` of type `nat`). There is one exit value (of type `nat`) and one local variable defined (`data` of type `nat`). The features added to basic state machines include:

- **Variables.** A state machine has number of typed data variables declared. Data variables define values which are passed to or from the state machine during a state transition. Each label on a transition consists of a gate name and a list of variables/values. As in LOTOS, the notation *!value* represents a value output by the state machine; the notation *?variable* represents a value read in by the state machine. The value of variables is persistent and does not change until a new value is read in for the variable. In figure 4.19 a value for variable `data` is input on the transition `get`; this value is then output on one of the transitions `put1` or `put2`.
- **Exit values.** A set of typed exit values can be defined. These values are returned by the state machine whenever it enters an exit state. In the example, whenever state `exit1` or `exit2` is entered the state machine will halt and output the value of `data`.
- **Conditional transitions.** A condition can be associated with each state transition. The transition can occur only if the condition evaluates to *true*. The condition can include typed data values as well as any declared variables. A condition on a transition is indicated by enclosing it in square brackets. In the example state machine, the transition `put2` is enabled only if the value of `data` is less than or equal to `const`; else the transition `put1` is enabled.



```

process nonDeterministicBuff[get,put1,put2] (const:nat):exit (nat) :=
  fsm[get,put1,put2] (const,init,0)
where
  process fsm[get,put1,put2]
    (const:nat ,state:state, Data:nat) : exit(nat) :=
      [state eq init] -> get ?Data_x:nat;
        fsm[get,put1,put2] (const,s1,Data_x)

      []
      [state eq s1] -> put2 !Data [Data le const];
        exit(Data)

      []
      [state eq s1] -> put2 !Data [Data le const];
        exit(Data)
        []
        put1 !Data [Data gt const];
          exit(Data)

  endproc (* fsm *)
endproc (*nonDeterministicBuff *)

```

Figure 4.19 Example of a state transition diagram with parameters and exit values.

- **Parameters.** Data parameters are passed to the state machine when it is invoked. These formal parameters can be used as constants within the state machine definition. The state machine `nonDeterministicBuff` has the parameter `const` passed to it.

4.2.3 Structural Decomposition and Behaviour

Structural decomposition allows a designer to identify components of a system, describe the behaviour to these components, and then decompose the components into sets of interconnected subcomponents. An example of structural decomposition was given in §4.1.1.2, where the components of the producer consumer design structure given in figure 4.1 were decomposed into subcomponents (figure 4.4).

However structural decomposition can cause a problem when comparing behaviour created at different levels of the structural design. In particular, two issues arise which are relevant to behavioural specification when decomposition of structure is performed:

- **Hidden interactions.** A designer can specify a behaviour B of a set of components using the interaction points (IPs *tick*, *tock*, *put*, *get* in the producer consumer structure of figure 4.1). When structure is decomposed, new interaction points are created (e.g., *bint*, *cint1*, *cint2*) and the designer can create a behavioural design B_D for the decomposed structure using these new IPs. The issue that arises is how to compare B and B_D in order to verify that they are compatible with each other when behaviour B_D contains interactions which are not included in B .
- **Decomposed components.** When creating a behavioural design B for a structure $\langle IP, C \rangle$, the designer describes the behaviour in terms of interactions, where every interaction is a triple $\langle i, c, V \rangle$ with c being a list of the components involved in the interaction. In the producer consumer structure for example, one interaction is $\langle put, [prod1, buff], [[5], []] \rangle$ (represented as

`put!prod1!buff!5` in LOTOS). When the components of $\langle IP, C \rangle$ are decomposed the designer creates a behavioural design B_D not in terms of the components of C but in terms of the subcomponents from which C is constructed; thus the interaction $\langle i, c, V \rangle$ in behaviour B will be specified as $\langle i, c', V \rangle$ in B_D where c' are the corresponding subcomponents of c . In the producer consumer example, the interaction $\langle put, [prod1, buff], [[5], []] \rangle$ seen at one level of the structural hierarchy becomes the interaction $\langle put, [prod1, b1], [[5], []] \rangle$ when viewed at another level of the structural hierarchy; when comparing behaviours across the different structural levels these two interactions must be considered the same interaction.

The problem of hidden interactions can be handled quite easily in LOTOS by using the ‘hiding operator’. Assuming that B and B_D are LOTOS expressions, comparison of behaviours can be done by comparing B to the expression:

`hide g_1, \dots, g_n in B_D`

where $\{g_1, \dots, g_n\}$ are the set of IPs hidden from B . For example in a resource oriented specification of the producer/consumer problem, assume that:

- The behaviours of `prod1`, `prod2`, `buff`, `cons1` and `cons2` (figure 4.1) are given by `p1`, `p2`, `b`, `c1` and `c2` respectively.
- The behaviours of `b1`, `b2`, `c11`, `c22`, `c21`, and `c21` (figure 4.4) are given by `b1`, `b2`, `c11`, `c22`, `c21`, and `c21` respectively.

The resource oriented specification of the system is given in figure 4.20. The resource oriented specification using the decomposed components is illustrated in figure 4.21. A verification can show that these specifications are equivalent, if the hidden interactions are removed.

The second problem, where the interactions in B involve the components of C while the interactions in B_D involve the subcomponents of C cannot be solved quite so elegantly using only the operators of LOTOS. In this case the solution is implemented at the semantic level, rather than a simple syntactic

```

P1 ||| default(prod1)
  ||
P2 ||| default(prod2)
  ||
B  ||| default(buff)
  ||
C1 ||| default(cons1)
  ||
C2 ||| default(cons2)

```

Figure 4.20 Resource oriented specification example.

```

P1 ||| default(prod1)
  ||
P2 ||| default(prod2)
  ||
hide bint in
  ( B1 ||| default(b1)
    ||
    B2 ||| default(b2)
  )
  ||
hide cint1 in
  ( C11 ||| default(c11)
    ||
    C12 ||| default(c12)
  )
  ||
hide cint2 in
  ( C21 ||| default(c21)
    ||
    C22 ||| default(c22)
  )

```

Figure 4.21 Resource oriented specification example with decomposed components.

manipulation of the LOTOS expression. Assume that B_D and B are LOTOS expressions with corresponding models $\mathfrak{M}(B_D)$ and $\mathfrak{M}(B)$. When verifying the design by comparing these two behavioural models it is necessary to perform a renaming on $\mathfrak{M}(B_D)$; for a specific component $c \in C$ all subcomponents of

c which are referenced in $\mathfrak{M}(B_D)$ must be replaced by the identifier c . Thus the interaction $\langle \text{put}, [\text{prod1}, \text{b1}], [[5], []] \rangle$ of the decomposed producer consumer problem will be replaced by the interaction $\langle \text{put}, [\text{prod1}, \text{buff}], [[5], []] \rangle$ when compared to behaviour at the higher level of structural decomposition.

For example in the producer/consumer problem, given the interaction sequence

```

tick !ext1 !prod1;
  put !prod1 !b1 !5;
    bint !b1 !b2 !5;
      get !b2 !cons1 !5

```

application of the hiding operator and renaming the internal components of `buff` will result in the following sequence (Note: `i` represents an internal hidden interaction):

```

tick !ext1 !prod1;
  put !prod1 !buff !5;
    i;
      get !buff !cons1 !5

```

Note that the components `b1` and `b2` have both been replaced with the component `buff`.

The component renaming can be done by defining a set of *structure renaming functions*, with one function defined for each node in the structure decomposition tree. Assume component c is decomposed into subcomponents $c_1, c_2 \dots c_n$. Then the structure renaming function $\Delta_c : \text{Components} \rightarrow \text{Components}$ associated with node c of the decomposition tree is defined as:

$$\Delta_c(k) = \begin{cases} c & \text{if } k \in \{c_1, c_2 \dots c_n\} \\ k & \text{otherwise} \end{cases}$$

Thus the structure renaming function Δ_{buff} for the buffer component in the producer consumer example is defined as follows:

$$\Delta_{buff}(x) = \begin{cases} buff & \text{if } x \in \{b1, b2\} \\ x & \text{if } x \notin \{b1, b2\} \end{cases}$$

4.2.4 Discussion

This section selected a behavioural specification language (LOTOS) and integrated this language with the structural design representation in order to represent behaviour in either a resource oriented or a slice oriented style.

A different approach to integrating behavioural design and structural design within a formal framework can be found in the work of Turner [80,81]. Turner identifies a set of *architectural properties* (similar to the structural design properties of this thesis). He then maps these architectural properties onto the characteristics of different formal languages such as LOTOS [46], Estelle [43], and SDL [15] in order to use these languages to represent behavioural design. Rather than assuming a concurrent system architecture as is done in this research, Turner uses the layered architecture of OSI [44] as the structural model and proceeds to show how the concepts in the layered architecture can be represented in the formal framework of the behavioural languages.

Another approach to formalizing behaviour and structure can be found in the work of Najm and Stefani [70,69]. Najm and Stefani define a language OL1 which can be used to specify the structure and behaviour of distributed systems. They allow for dynamic structure within their representation where components (objects in their terminology) and interconnections are created during the life of the system. The OL1 language is based on process algebras and Najm and Stefani provide a transformation from an OL1 specification into a LOTOS specification. In terms of structural representation, the main extension provided by this thesis is the hierarchical structural representation whereby components can be decomposed into

subcomponents. Najm and Stefani also make the assumption that an intensional description of behaviour is always done using a resource oriented style.

MachineCharts [10,12] defines behaviour of a structure using a combination of resource oriented and slice oriented approaches. Resource oriented specifications for primitive components are defined using a combination of state machines, graphical icons, and high level coding. Slice oriented behaviour is defined using event sequences or sequences of extended rendezvous. These behaviours can be defined visually using an informal timeline notation similar to the notation of this thesis.

Raddle/Verdi defines behaviour using a notation based on Petri nets [73]. Only the behaviour of primitive components can be defined. There is no provision for slice oriented behaviour specifications.

TELOS [78,77] specifies behaviour using a combination of state transition systems and high level coding. Only the behaviour of primitive components can be specified. There is no provision for slice oriented behaviour specifications.

4.3 Chapter Summary

This chapter has developed a means for representing the structural and behavioural designs of a concurrent system.

The structural design consists of a set of components which are interconnected by means of a set of interconnection points (IPs). Each IP represents a place where an interaction, defined as a synchronous rendezvous, can occur between two or more components. During the interaction, data can be exchanged between the components involved in the interaction. A visual notation is used in order to capture the structural design.

The behavioural design is represented using the LOTOS specification language. Using the abstract data types of the language, LOTOS is integrated with the structural design representation. It is then shown how LOTOS can be used

to represent the behaviour in either a resource oriented or a slice style specification. In order to assist a designer in capturing behavioural design information, graphical notations are developed.

Chapter 5: Formalizing Slice Techniques

In order to support a design process, it is necessary to formalize the concept of a *slice* and to develop a theory which allows slice expressions to be used during the development of concurrent systems. Section 3.1 proposed a set of design steps using slices. This chapter develops some theoretical results about slice expressions which can be used to support the design steps. (Chapters 6 and Appendix D will describe an example and a larger case study which show how the theory developed within this chapter can be applied.)

The design steps of interest to this research were identified in §3.1 as being the following:

1. *Perform the structural design.*
2. *Explore the behaviour of the structural design through slice expressions.*
3. *Develop component behaviour expressions.*
4. *Verify the component behaviour specifications.*
5. *Decompose complex components.*

Sections 5.1 to 5.4 show how slice expressions which are formally represented as LOTOS behaviour expressions can be used to support this set of design steps.

In order to support the above steps, a theory of slice expressions must be developed which includes the following elements:

- *Relations between behaviour expressions.* There are numerous examples in the design steps where a formal relationship between behaviour expressions can be defined and used as a basis for testing and verification. For example, as slice expressions are refined to represent more complex behaviour, it is possible to verify that the refinement does not violate any previously specified behaviour. Section §5.1 identifies the relation *ext* as the basis for comparing and verifying behaviour expressions.

- *Developing component specifications.* Once a designer has explored behaviour through slice expressions, specifications for individual components can be developed. Methods for deriving component behaviour from slice expressions are explored in §5.2.
- *Design analysis.* Slices provide a means by which a design can be analyzed for critical races. Section §5.3 describes how slices can assist in detecting such race conditions.
- *Verifying component and system behaviour using slices.* Slice expressions provide a set of test cases which can be used to verify the correct behaviour of an individual component specification or to verify the behaviour of a resource oriented system specification. Section §5.4 describes how each slice expression can be considered as a set of test cases for a component or for a set of components.

5.1 Relations Involving Slice Expressions

During behavioural design, a designer creates a number of behaviour expressions which capture different aspects of overall system behaviour. Each of these expressions will be related to each other in some way. For example:

1. **Relations between slice expressions.** Slice expressions are constructed in an incremental fashion, beginning with simple slices showing basic functionality and then refining these slice expressions to define more complex behaviour patterns. If S_1 is a simple slice expression and S_2 is refinement of S_1 , then S_2 must provide all the functionality of S_1 , but may provide more; thus S_2 *extends*¹ S_1 .
2. **Relations between slice expressions and resource oriented expressions.** A slice expression is a partial specification of behaviour which describes some functionality which must be provided by the system. A resource oriented

¹ As noted earlier, the word *extension* has two distinct meanings in this thesis. *Slice extension* is a technique for refining behaviour. The *extension* relation (signified by *ext*) is a relation between transition systems.

specification is a complete specification of behaviour which describes the behaviour of each component and then composes the component behaviours to represent the system behaviour. Thus if S_1 is a slice expression of behaviour, and $B_{\langle IP, C \rangle}$ is a resource oriented specification of behaviour, $B_{\langle IP, C \rangle}$ must provide all the functionality of S_1 , but may provide more; therefore $B_{\langle IP, C \rangle}$ *extends* S_1 .

What precisely is meant by one behaviour *extending* another behaviour? Behaviours are defined entirely in terms of observations which can be made at intercomponent IPs. The two things which are observable at an IP are (see §4.2.1.1): an interaction occurring at the IP; and the set of interactions which are offered by each component at the IP. Defining one behaviour as *extending* another behaviour must be done in terms of these two observable properties. Thus, saying that behaviour S_2 *extends* behaviour S_1 can be defined informally as:

- S_2 *must provide at least as much behaviour as* S_1 . If behaviour S_1 is capable of engaging in trace t , then behaviour S_2 must also be capable of engaging in trace t .
- S_2 *cannot refuse more than* S_1 . If t is a trace of both S_1 and S_2 , and after engaging in t behaviour S_2 may refuse to engage in the interactions of set A , then S_1 after executing trace t may refuse to engage in any interaction of A .

The relation *ext* which satisfies the above properties is formally defined in §2.3.3.

Using the relation *ext*, a more precise definition of the ‘meaning’ of a slice expression can be given. In particular, points 1 and 2 of page 114 can be formalized by the following assumptions.

Assumption 1. If S_1 and S_2 are slice expressions and S_2 is a refinement of S_1 then S_2 *ext* S_1 .

Assumption 2. If $B_{\langle IP, C \rangle}$ is a resource oriented behaviour specification of a system and S is a slice expression for the same system, then $B_{\langle IP, C \rangle}$ is correct only if $B_{\langle IP, C \rangle} \text{ ext } S$.

Section 5.1.1 briefly discusses other possible relations which could have been used as a basis for defining the meaning of a slice expression, and why the *ext* relation was considered the most appropriate.

5.1.1 Other Possible Relations for Defining Slices

The meaning of a slice expression and its relation to the resource oriented behaviour expression was captured by the relation *ext* (extension). This relation is based on the concept of *failures*. There are other relations which could be chosen as the basis for defining the constraints imposed by a slice expression. Two such relations will be discussed here: one based on *simulation* and one based on *traces*.

5.1.1.1 An interpretation based on *simulation*

A stronger relationship than *ext* is one based on *simulation*. The relation usually defined in the literature is one called *bisimulation* where two transition systems are *bisimilar* if they are capable of simulating each other.

Definition: Bisimulation [65,3]

Given transition systems $B_1 = \langle \Sigma_1, I_1, T_1, init_1 \rangle$ and $B_2 = \langle \Sigma_2, I_2, T_2, init_2 \rangle$, a relation \sim between states of B_1 and B_2 is a *bisimulation* if for any pair of states s_1, s_2 such that $s_1 \sim s_2$ and for any sequence t of observable interactions:

- i. whenever $s_1 \xrightarrow{t} s'_1$ then for some $s'_2 : s_2 \xrightarrow{t} s'_2$ and $s'_1 \sim s'_2$
- ii. whenever $s_2 \xrightarrow{t} s'_2$ then for some $s'_1 : s_1 \xrightarrow{t} s'_1$ and $s'_1 \sim s'_2$

The relation *bisimulation* is stronger than a relation based on *failures* in that $S1 \sim S2$ implies that $failures(S1) = failures(S2)$ but not vice versa.

The difference between the two interpretations is illustrated by the example of figure 5.2. This figure shows two transition systems; the transition systems represent two different behavioural specifications of the component of figure 5.1. It can easily be verified that $failures(S2) = failures(S1)$. However these transition systems have a very distinct behavioural difference. In both $S1$ and $S2$ the component will engage in interaction sequence $a.b$ and then either offer interaction x and refuse y , or offer y and refuse x . The critical difference between the two is when the decision is made as to which of interactions x or y is to be offered. In behaviour $S1$ the decision is made when engaging in the b interaction; in behaviour $S2$ the decision is made when engaging in the a interaction.

Although it is true that $S2$ and $S1$ have the same *failures*, it is not true that $S2$ and $S1$ are *bisimilar*. To see this note that $S1$ after engaging in interaction a moves to state T ; $S2$ after engaging in interaction a moves to state $T1$ or $T2$. Neither $T1$ nor $T2$ is a simulation of T ; therefore it is not true that $S2$ *bisimulate* $S1$.

The *bisimulation* relation was not chosen as the basis for relating slice expressions to system behaviour since it is too strong a relation. As described in §4.2.1.1 a behaviour specification defines the possible observations which can be seen at the probes located at the IPs. Assuming that the transition systems of figure 5.2 refer to the component structure of figure 5.1 there is no observable difference between the behaviour of $S1$ and $S2$ which can be seen simply by watching the IP probes. As a behavioural specification we wish to consider the two specifications of figure 5.2 equivalent; *bisimulation* however makes a distinction between them.

5.1.1.2 An interpretation based on *traces*

Rather than basing the relation between the slice expression S and the system expression $B_{\langle IP, C \rangle}$ on *failures* it is possible to base the relation on *traces*. In this case the only requirement would be for $traces(B_{\langle IP, C \rangle}) \supseteq traces(S)$.

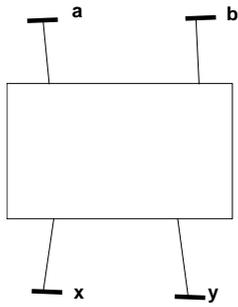


Figure 5.1 Example structure to illustrate problem with the bisimulation relation.

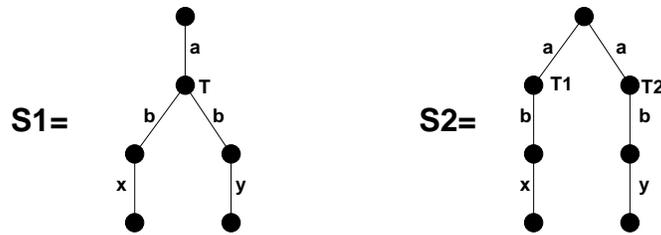


Figure 5.2 Transition systems which are failure equivalent, but which are not bisimilar.

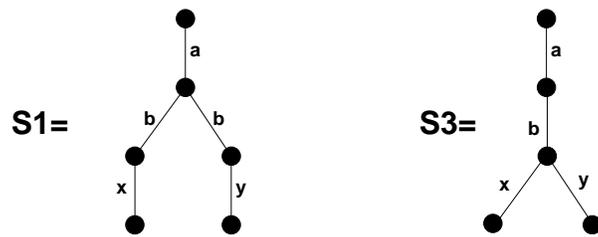


Figure 5.3 Comparison of refusals and traces as a basis for relating slice expressions and system behaviour expressions.

The problem with using *traces* rather than *failures* as the basis for the relation is that *traces* alone cannot differentiate between component behaviours which are observably different as seen at the IP probes. This is illustrated by the example of figure 5.3. The two transition systems $S1$ and $S3$ have identical trace sets. However assume that an observer has seen the system engage in trace $a.b$. If the component behaviour is given by $S1$ then the observer checking the probe at IP x may see the event x being offered, or may see the event x being refused; which observation is made depends on an earlier nondeterministic choice made by the component during interaction b . If however the component's behaviour is given by $S3$ then after $a.b$ the observer will always see the event x being offered. Using only *traces* there is no way to differentiate these two behaviours even though they lead to different observations.

5.2 Developing Component Behaviour Specifications

Once a designer has explored system behaviour through slices, one of the design steps involves developing individual component specifications. This section investigates how slices can assist a designer in developing the individual component specifications.

A slice style specification defines behaviour patterns which cross many components. Implementation of the components requires that there exists a behavioural specification for each individual component. Given a slice expression, what information does the slice expression imply about the behaviour of an individual component? Can the designer use the slice expressions as a starting point for generating the individual component specifications? The problem is illustrated in figure 5.4. Beginning with the slice expression $S1$ on the left, a designer wishes to develop the component behavioural specification B_c for the component c .

Slice expressions in this research are represented as LOTOS behaviour expressions. Every LOTOS behaviour expression defines a transition system [46]. Therefore given a LOTOS behaviour expression S which represents a slice style

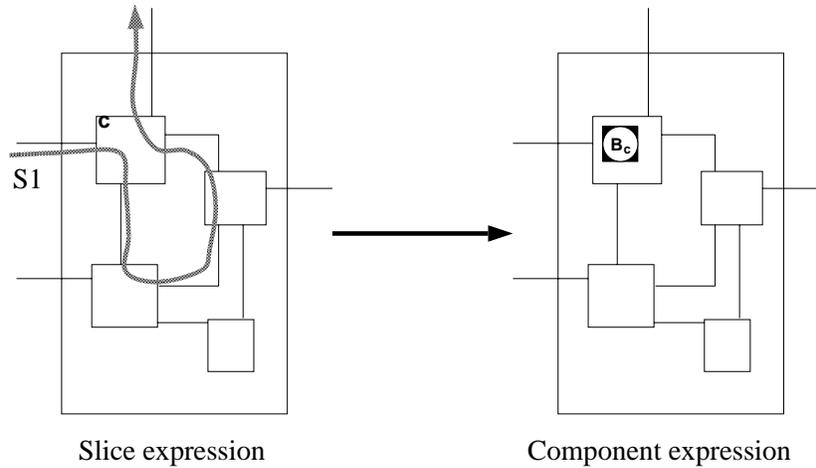


Figure 5.4 Extracting component expressions from slice expressions.

specification, there are two approaches that can be used to generate a LOTOS expression B_c which specifies the behaviour of component c implied by S :

1. Define a symbolic transformation ϕ_c which maps LOTOS specifications to LOTOS specifications; the specification $\phi_c(S)$ represents the behaviour of component c . This approach is illustrated in figure 5.5 and is called the *syntactic approach*.
2. Generate the transition system model $\mathfrak{M}(S)$. From the transition system $\mathfrak{M}(S)$ extract a transition system \mathfrak{M}_c which represents the behaviour of component c . Working backwards from \mathfrak{M}_c , construct a LOTOS behaviour expression B_c which represents the behaviour of component c and for which $\mathfrak{M}(B_c)$ satisfies any requirements of \mathfrak{M}_c . This approach is illustrated in figure 5.6 and is called the *semantic approach*.

5.2.1 The Syntactic Approach

A slice specification S written in some language L describes the behaviour of a set of interconnected components. Given a specific component c of this

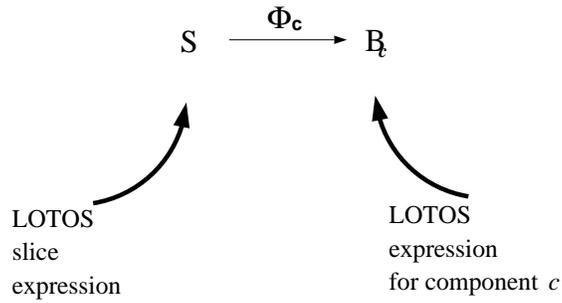


Figure 5.5 Syntactic approach to extracting component behaviour from slice expression.

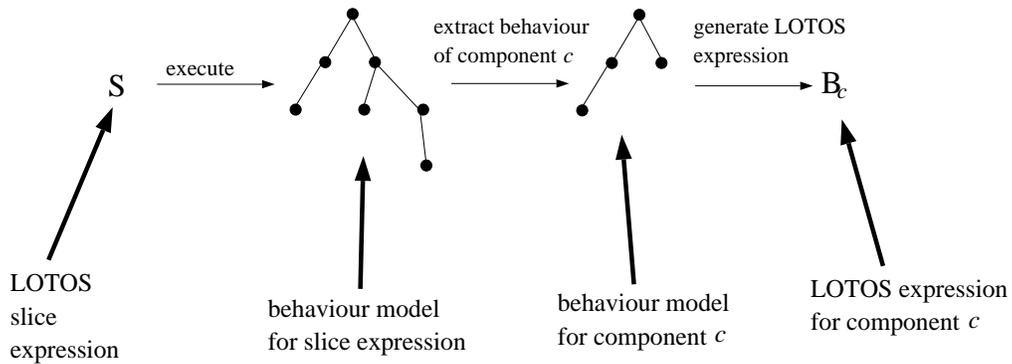


Figure 5.6 Semantic approach to extracting component behaviour from slice expression.

component set, one method of determining the behaviour of c implied by slice S is to define a syntactic transform which will convert the slice expression S into the component expression S_c . This section introduces the ϕ_c transform which converts a LOTOS slice expression to the LOTOS component expression which describes the behaviour of component c . It should be noted when reading this section that the syntactic transform developed does not exactly capture the requirements of the individual components; as discussed at the end of the section there are difficulties in correctly extracting a component expression from the slice expression. However, the component behaviour extracted can be assumed to

form a basis from which the complete and correct component specification can be developed.

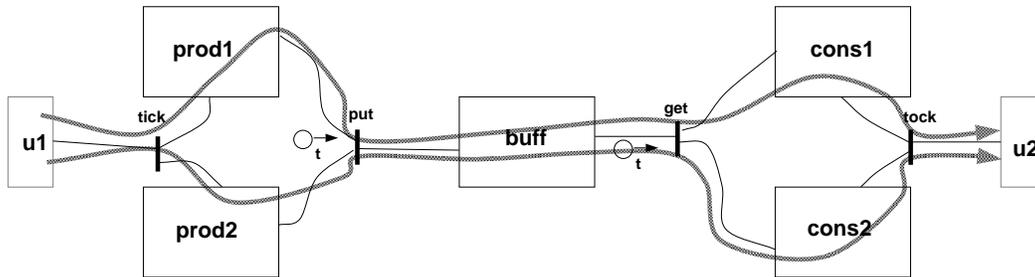
The producer/consumer problem will be used as an example to illustrate the ϕ_c transform; figure 5.7 shows the design structure and a corresponding slice expression. Note that although the structural design of this example is the same structural design of the producer/consumer example of Chapter 4, the behavioural designs are different.

The expression `slice` is constructed from two invocations of the process `slice1` combined in parallel (lines 2 to 4). The invocation at line 2 shows a value being passed from `prod1` to `cons1`; the invocation at line 4 shows a value being passed from `prod2` to `cons2`.

The process `slice1` defined at lines 6 through 14 shows the behaviour of the system when a value is being passed from producer `p` to buffer `b` and then to consumer `c`. The producer engages in a `tick` interaction (line 7) and then passes some value `x` of type `t` to the buffer by a `put` interaction (line 8). The choice operator at line 12 then chooses between two possible behaviours depending on whether the condition `cond(x)` evaluates to `true`. If `cond(x)` evaluates to `true` (line 9) then the value `x` is passed to the consumer by a `get` interaction (line 9), and the consumer issues a `tock` (line 10). If `cond(x)` does not evaluate to `true` then nothing happens (line 13).

The steps involved in the ϕ_c transform are the following:

1. *Remove events not involving component c .* The first step in extracting the component behaviour of c from slice S is to remove all interactions from S which do not involve component c . The steps involved in actually removing these interactions is formally defined by the `phi` function of Appendix E and described in more detail in §5.2.1.1.
2. *Simplify the component expression.* Once the interactions not involving component c have been removed, the resulting behaviour expression can often



specification slice

```

1 behaviour
2   slicel[...] (prod1, buff, cons1)
3   |||
4   slicel[...] (prod2, buff, cons2)
5
6   where
7     process slicel[...] (p,b,c: component) : exit :=
8       tick !u1 !p;
9       put !p !b ?X:t;
10      ([cond(X)] -> get !b !c !X;
11       tock !c !u2;
12       stop)
13      ([not cond(X)] -> stop)
14   endproc

```

Figure 5.7 Slice expression for the consumer/producer structure.

be simplified by means of term rewriting. The term rewriting rules which are applied are defined in the table of figure 5.11.

3. *Generalize specification to take into account data flow and component anonymity.* Slice expressions do not take into account data flow during an interaction whereby a data value is passed from one component to another; a slice expression will simply indicate that a specific data value is agreed upon by both components. Nor do slices take into account ‘anonymous components’ where, for example, in a client/server relation a server will accept a request from any available client. In each slice expression the components involved in each interaction are fixed; in an implementation a server may view the clients as being anonymous and be willing to accept requests from any client making a request. Component specifications can be extended to include data flow information and component anonymity. The method for performing such a generalization is for the designer to identify expressions in the LOTOS component specification which represent data flow items or anonymous components, and then to apply the data extension function defined in §5.2.1.3.

For the producer/consumer problem, the three steps can be applied to the expression `slice` in order to extract the behaviour of component `cons1`. Applying the three steps of the ϕ_{cons1} transform is done as follows:

1. The LOTOS expression `cons1_A` of figure 5.8 is the behaviour of component `cons1` which was derived from the specification of figure 5.7 by application of the transform `phi`. This expression is similar to the expression `slice` of figure 5.7 with the following modifications:
 - The interactions not involving component `cons1` (i.e., `tick`, `put`) have been removed.
 - Since the variable `x` is not defined by `cons1` at line 9 where the condition must be evaluated, the Boolean condition `cond(x)` of expression `slice` has been removed.

- The process `slice1` has resulted in two process definitions, `slice1_ca` and `slice1_cb`. The process definitions appear at lines 6 and 13 of specification `cons1_A`; the process instantiations are at lines 2 and 4.
2. By using the term rewriting rules defined in figure 5.11, the expression `cons1_A` can be simplified to the expression `cons1_B` defined in figure 5.9. This simplification removes the trivial process `slice1_cb` and removes the trivial choice `stop` at line 11 of specification `cons1_A`
 3. Finally, the designer can generalize the behaviour of `cons1` by making the external component with which the consumer interacts anonymous to the consumer. If the external component is anonymous, the consumer will not specify before the interaction occurs with which component it must interact; it will interact with any component on the other side of the `tock` IP. The result is the specification `cons1_c` of figure 5.10, where at line 6 the expression `!u2` has been replaced by `?y:component`.

Since each slice S is an incomplete specification of the behaviour of the system, the specification $\phi_c(S)$ is an incomplete specification of the behaviour of component c . What exactly is the relation between $\phi_c(S)$ and the behaviour of component c ? If t is a trace of S , and t_c is the trace t restricted to the interactions involving component c , it can be proven that t_c must be a trace of $\phi_c(S)$ (theorem 1). Thus, we can be relatively confident that in some sense the expression $\phi_c(S)$ does capture the behaviour of component c which is implied by S .

Theorem: 1

$$\forall t \in \text{traces}(S) \bullet t \upharpoonright I(c) \in \text{traces}(\phi_c(S))$$

Unfortunately, given that B_c represents a complete specification of the behaviour of component c , there is no simple relation which can be defined between $\phi_c(S)$ and B_c . The reasons for this are discussed further in §5.2.1.4. However, it is always possible to begin with $\phi_c(S)$ as a preliminary specification of com-

```

(*Behaviour of a consumer*)
specification cons1_A
1  behaviour
2    slicel_ca[...] (buff,cons1)
3    |||
4    slicel_cb[...]

5  where
6    process slicel_ca[...] (b,c : component) : exit :=
7      (get !b !c ?X:t;
8        tock !c !u2;
9        stop)
10     []
11     (stop)
12  endproc

13  process slicel_cb[...] : exit :=
14    stop
15    []
16    stop
17  endproc

```

Figure 5.8 Behaviour of component **cons1** extracted from slice expression by **phi**.

ponent c , and then *extend* $\phi_c(S)$ into the complete specification B_c . Thus, the following assumption will be made:

Assumption 3.

B_c is a correct behaviour specification of component c only if $B_c \text{ ext } \phi_c(S)$.

The process for developing component expression B_c from slice expression S can then be given as:

1. For each slice expression S generate the behaviour expression $\phi_c(S)$.
2. Explore the behaviour $\phi_c(S)$ in order to learn about the required behaviour of c .
3. By assumption 3, *extend* the expression $\phi_c(S)$ into the complete behaviour specification B_c .

```

(*Simplified behaviour of consumer*)
specification cons1_B

1  behaviour
2  slice1_ca[...] (buff, cons1)

3  where
4  process slice1_ca[...] (b,c : component) : exit :=
5  get !b !c ?X:t;
6  tock !c !u2;
7  stop
8  endproc

```

Figure 5.9 Simplified behaviour of component **cons1**

```

(*Consumer behaviour with external component anonymous*)
specification cons1_C

1  behaviour
2  slice1_ca[...] (buff, cons1)

3  where
4  process slice_ca[...] (b,c : component) : exit :=
5  get !b !c ?X:t;
6  tock !c ?Y:component;
7  stop
8  endproc

```

Figure 5.10 Component behaviour of **cons1** with external component 'anonymous'.

$B [] \text{stop}$	$\rightarrow B$
$B \text{stop}$ and B does not exit	$\rightarrow B$
$B \text{stop}$	$\rightarrow \text{stop}$
$[p] \rightarrow \text{stop}$	$\rightarrow \text{stop}$
$B [] B$	$\rightarrow B$
$B \text{exit}$	$\rightarrow B$
$\text{exit} \text{exit}$	$\rightarrow \text{exit}$
$P \text{ where } P := \text{stop}$	$\rightarrow \text{stop}$
$P \text{ where } P := \text{exit}$	$\rightarrow \text{exit}$
$(B_1 [] B_2) [] B_3$	$\leftrightarrow B_1 [] (B_2 [] B_3)$
$(B_1 B_2) B_3$	$\leftrightarrow B_1 (B_2 B_3)$
$(B_1 B_2) B_3$	$\leftrightarrow B_1 (B_2 B_3)$

Figure 5.11 Term rewriting used to simplify component expressions.

The following section provides further detail regarding the ϕ_c transform.

5.2.1.1 Description of the \mathbf{phi}_c Transform

In order to derive $\phi_c(S)$ from S , it is necessary to remove from S all interactions which do not involve component c . Removing these interactions can be partially automated by applying the algorithm \mathbf{phi} as defined in Appendix E. Given a slice expression S , the function \mathbf{phi} returns an expression with all interactions not involving component c removed. The \mathbf{phi} algorithm is defined recursively: to compute $\mathbf{phi}(s)$ apply the \mathbf{phi} function to the subexpressions of s and then compute $\mathbf{phi}(s)$.

The transformation has four main types of rules which it applies to deal with the following constructs:

- The action prefix operator.
- Operators for combining behaviour expressions.
- Boolean conditions.
- Process definitions and instantiations.

Action prefix operator.

The LOTOS action prefix operator ‘;’ is of the form $\mathbf{a};\mathbf{B}$ and represents the system engaging in interaction \mathbf{a} and then behaving like \mathbf{B} . To transform an expression of this form, the \mathbf{phi} function must determine whether \mathbf{a} represents an interaction involving component c ; if not, references to \mathbf{a} are eliminated and the transform recursively applied to \mathbf{B} . If \mathbf{a} represents an interaction involving component c then \mathbf{a} is retained. Thus,

$$\mathbf{phi}(\mathbf{a};\mathbf{B}) = \begin{cases} \mathbf{a};\mathbf{phi}(\mathbf{B}) & \text{if component } c \text{ is involved in interaction } a \\ \mathbf{phi}(\mathbf{B}) & \text{otherwise} \end{cases}$$

Component names are represented as variables. Therefore in order to determine whether a specific interaction within slice S involves component c it may be necessary to evaluate data expressions involving variables. For this reason, in some cases it may not be possible to completely automate the \mathbf{phi} transform and

some user guidance may be required to determine whether a variable can evaluate to a specific component identifier.

Since LOTOS permits variable declarations to be included as part of the event, the `phi` transform is complicated by the fact that removing the event `a` may also remove variable declarations. This is handled by recording all variable declarations which are removed when an event is removed; these declarations are recreated when the variable is next used.

The hidden interaction `i` is considered to be an interaction involving every component.

Behaviour expression operators.

If the transform is applied to an operator other than the action prefix operator, it is defined so that `phi` distributes over the operator. For example if

$$s = s1 ||| s2$$

then

$$\text{phi}(s1 ||| s2) = \text{phi}(s1) ||| \text{phi}(s2)$$

Boolean conditions.

There are a number of places in LOTOS where a Boolean condition can be used to control behaviour. These Boolean conditions are computed based on variables and constants of the LOTOS expression. When transforming a slice expression to a component expression, the transform must try to determine which of the components involved in the slice is responsible for enforcing the condition. The rule applied by the `phi` transform is the following: any component which has sufficient information to compute the Boolean condition maintains the condition as part of its behaviour expression; a component which does not have sufficient information to compute the Boolean condition assumes the condition to be *true* and the condition is removed from its behaviour expression. A component has sufficient information to compute the condition if and only if all variables of the

condition have been instantiated. By the semantics of LOTOS, a variable can be considered as being instantiated at the point where it is declared.

For example, the LOTOS expression $[p] \rightarrow B$ can be interpreted as “if p evaluates to *true* then behave like B ”. The phi transform must determine whether component c has sufficient information to evaluate condition p , i.e., are all variables in p declared. If no, then the condition is dropped and the resulting expression is $\text{phi}(B)$; if yes, then the condition is retained and the resulting expression is $[p] \rightarrow \text{phi}(B)$. Thus

$$\text{phi}([p] \rightarrow S) = \begin{cases} [p] \rightarrow \text{phi}(S) & \text{if } p \text{ can be evaluated} \\ \text{phi}(S) & \text{otherwise} \end{cases}$$

Process definitions and instantiations.

A LOTOS specification includes process definitions where a process is a parameterized behaviour expression. A process can be instantiated within a behaviour expression by providing actual parameters to replace the formal parameters of the process definition. In order to apply the phi transform both the process definition and the process instantiation must be transformed.

A process definition in LOTOS takes the following form:

$$P[g_1 \dots g_n](t_1 \dots t_m) := B$$

where P is the process name, $g_1 \dots g_n$ are the formal gate name parameters, and $t_1 \dots t_m$ are variable declarations defining the formal data parameters. The behaviour expression B is the body of the process definition. In order to apply the phi transform to the process definition we would like to be able to distribute the transform through the process definition so that:

$$\text{phi}_c(P[g_1 \dots g_n](t_1 \dots t_m) := B) = (P[g_1 \dots g_n](t_1 \dots t_m) := \text{phi}_c(B))$$

The problem is somewhat more complex than this however for two reasons:

1. Some of the data parameters $t_1 \dots t_m$ may be undefined at the point of process invocation. The solution is for the phi transform to remove the undefined variables from the parameter list and reinsert appropriate data definitions into the body of the process definition.

2. In order to compute $\text{phi}(\mathbf{B})$ the specific components involved in each interaction of \mathbf{B} must be known; the component identifiers may be computed from the data parameters $t_1 \dots t_m$. These values may not be known until invocation of the process. To handle this, the transform determines which data parameters are used to compute component identifiers. These parameters can then be used to determine whether a new process needs to be defined, or whether the appropriate process has already been defined. Deciding whether a new process definition is needed may require a decision to be made by the designer.

For example, assume that a process is defined as follows:

```

process P[ip] (c1,c2:Component) noexit :=
    ip !c1 !c2;
    stop
endproc

```

Applying phi to the process body will result in the expression $\text{ip}!c1!c2;\text{stop}$ or the expression stop depending on whether component c is one of the participants in the interaction $\text{ip}!c1!c2$. However this cannot be determined until the process is invoked and the parameters $c1$ and $c2$ are instantiated.

To overcome this problem, for each process definition P the phi transform creates a set of process definitions P_{c1}, \dots, P_{ck} . Each time a process instantiation is encountered, the phi transform must determine whether a new process definition is required (such a decision may require user assistance). If yes, a new process identifier is created and the process defined.

5.2.1.2 Simplifying the Component Expressions

Once the component behaviour expressions have been extracted from the slice expression S by means of the function phi , there are often a number of simplifications which can be made to the resulting component expression. This

can involve removing the trivial behaviour expressions `stop` or `exit`, or simplifying expressions of the form $\mathbf{B} \parallel \mathbf{B}$. The table of figure 5.11 provides a list of rewrite rules which can be applied to simplify a component expression.

In order for the rewrite rules to be safely applied to the component expressions, it must be shown that they can be applied to a component expression without changing the *failures* of the resource oriented system specification. In other words, given the following:

- For each component $c \in C$ the behaviour of component c is given by expression B_c .

- The resource oriented specification $B_{\langle IP, C \rangle}$ is defined as:

$$\parallel_{c \in C} (B_c \parallel \text{default}(c)).$$

- For each $c \in C$ the expression B'_c is derived from B_c by application of the rewrite rules of figure 5.11.

- The resource oriented specification $B'_{\langle IP, C \rangle}$ is defined as:

$$\parallel_{c \in C} (B'_c \parallel \text{default}(c)).$$

It must be shown that:

$$\text{failures}(B_{\langle IP, C \rangle}) = \text{failures}(B'_{\langle IP, C \rangle})$$

This can be shown by noting that the rewrite rules of figure 5.11 are congruence rules as defined in the LOTOS standard [46].

5.2.1.3 Generalize specification to take into account data flow and component anonymity

Including data flow and anonymous components into a component specifications requires performing an operation called “data extension”. The data extension operation is used to remove a specific data expression from a behaviour expression, and replace it with a variable. For example:

- To indicate that a specific data value \mathbf{a} of type `type` in a slice expression is an input value to component c during the interaction, the token `!a` must be

replaced by $?x:\text{type}$ where the value a is input to the component; elsewhere in the behaviour expression the value a must be replaced by the variable x .

- To indicate that component c will interact with any component during interaction e , interactions in the slice of the form $e!c!c2$ must be replaced by $e!c?Y:\text{component}$; any occurrence of $c2$ in a data expression must be replaced by variable y .

The steps involved in a data extension are the following:

1. Identify a data expression e which is a data parameter of an event in the form $!e$.
2. ‘Mark’ all instances of the expression e which are used by the component either to output values to other components or to compute other expressions.
3. Transform the expression as follows:
 - Convert the first instance of $!e$ to $?x:\text{type}$, where x is a unique variable name and type is the sort of expression e .
 - Convert all marked instances of e to x .

5.2.1.4 Difficulties with the Syntactic Transform

Given that $\phi_c(S)$ is the behaviour of component c derived from slice expression S and that B_c is the complete specification of the behaviour of c , what is the relation between $\phi_c(S)$ and B_c ? Assumption 3 (page 126) required that $B_c \text{ ext } \phi_c(S)$. Rather than making an assumption regarding this relation, it is desirable to prove a theorem which defines the relation between $\phi_c(S)$ and B_c . However, there are a number of reasons why no simple relation can be proven:

- **Nondeterminism.** When nondeterminism is encountered in a slice expression, it means that the slice behaviour can progress in a number of possible ways; in which particular way it progresses is left as a nondeterministic choice. How is this choice made when the slice specifies the behaviour of

a set of components? Probably what the designer had in mind was that one specific component of the set is nondeterministic; the other components are deterministic and must be able to progress along any of the behaviour paths selected by the nondeterministic component. Unfortunately when nondeterminism is encountered in a slice expression there is insufficient information in the expression to determine which component the designer intended to be nondeterministic. The assumption made by ϕ_c is that all components are nondeterministic.

- **Choice.** A similar problem to nondeterminism occurs when using the LOTOS choice operator ‘ \square ’ in a slice expression. A slice expression of the form $s_1 \square s_2$ means that a choice is to be made between behaviour s_1 and behaviour s_2 . The probable intent of the designer is that one specific component of the slice makes the choice; the other components must then be able to accommodate either choice. Unfortunately it is not always explicit from the slice expression which component is making the choice and it is possible for the ϕ_c transform to introduce nondeterminism into the other components where none was intended.
- **Boolean conditions.** The ϕ_c transform assumes that any Boolean condition of the slice S which cannot be evaluated by c must be assumed to be *true*. This leads to behaviours of $\phi_c(S)$ which are not necessarily implied by S .

The nondeterminism is a problem in both the syntactic and semantic approach to determining component behaviour and is discussed further in §5.2.3. The problems of choice and Boolean conditions are discussed below.

Choice operator. The problem of the LOTOS choice operator is one of trying to maintain ϕ_c as a ‘context free’ transform so that $\phi_c(S_1 \square S_2) = \phi_c(S_1) \square \phi_c(S_2)$. Unfortunately applying ϕ_c to a LOTOS behaviour expression containing the choice operator can introduce behaviour into a component expression that is not in the original slice expression. This is illustrated by the example of figure 5.12

which shows a structural design consisting of three components, one of which is external. Underneath the structure diagram are four behaviour expressions and their corresponding transition system models. The first expression is a slice illustrating the behaviour where the system chooses between interaction \mathbf{a} and \mathbf{b} and then depending on the choice engages in $\mathbf{q};\mathbf{p};\mathbf{stop}$ or just in $\mathbf{q};\mathbf{stop}$. Clearly the intention of the designer is that component \mathbf{x} makes the choice as to which behaviour the system is to engage in after one of the interactions \mathbf{a} or \mathbf{b} .

The second behaviour expression is the result of applying the transform ϕ_Y to the slice expression. As can be seen, the specification for component \mathbf{y} is nondeterministic. It engages in event \mathbf{q} and then makes a nondeterministic choice as to whether or not to engage in \mathbf{p} . Thus when the behaviour expressions $\phi_Y(S)$ and $\phi_X(S)$ are composed (third behaviour expression in figure 5.12), the resulting behaviour expression can stop after engaging in $\mathbf{a};\mathbf{q}$; in the slice expression, after engaging in $\mathbf{a};\mathbf{q}$ the system must offer interaction \mathbf{p} . A ‘corrected’ specification of component \mathbf{y} is shown as the last behaviour expression of figure 5.12. However the corrected specification of component \mathbf{y} was not generated using the transform.

Boolean conditions. In order for a Boolean condition in a slice expression to have meaning it must be computable by at least one component of the system. The ϕ_c transform assumes that the condition is evaluated by every component in which all variables of the condition have been instantiated; for other components the condition is assumed to be *true*.

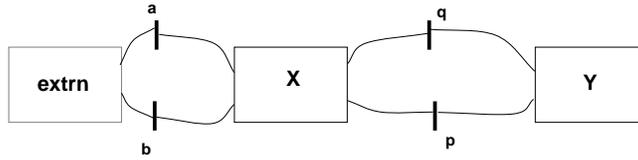
Assuming Boolean conditions *true* can lead to behaviour being included in the expression $\phi_c(S)$ which is not directly implied by slice S . For example, assume a slice expression of the form:

$$[\mathbf{x} \text{ eq } \mathbf{x}+1] \text{->B}$$

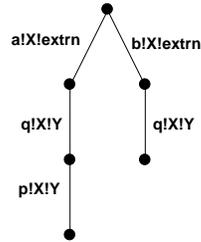
If the transform ϕ_c is applied when \mathbf{x} has not been instantiated, then

$$\phi_c([\mathbf{x} \text{ eq } \mathbf{x}+1] \text{->B}) = \phi_c(\mathbf{B})$$

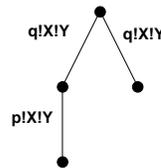
However since the condition $[\mathbf{x} \text{ eq } \mathbf{x}+1]$ is always *false*, the correct transformation



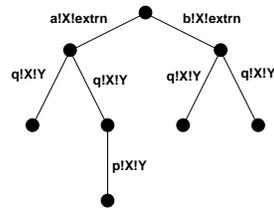
(* slice expression *)
 (a!X!extrn; q!X!Y; p!X!Y; stop)
 []
 (b!X!extrn; q!X!Y; stop)



(* Component expression for Y
 derived from slice *)
 (q!X!Y; p!X!Y; stop)
 []
 (q!X!Y; stop)



(*Composition of component expressions*)
 (*specification for component X*)
 ((a!X!extrn; q!X!Y; p!X!Y; stop)
 []
 (b!X!extrn; q!X!Y; stop))) ||| default[..](X)
 ||
 (*specification for component Y*)
 (((q!X!Y; p!X!Y; stop)
 []
 (q!X!Y; stop))) ||| default[..](Y)
 ||
 (*specification of external component*)
 B_extrn ||| default[..](extrn)



(*Corrected expression for
 component Y *)
 (q!X!Y; p!X!Y; stop)



Figure 5.12 Slice expression using choice operator.

should be:

$$\phi_c([\mathbf{x} \text{ eq } \mathbf{x}+1] \rightarrow \mathbf{B}) = \text{stop}$$

Unfortunately there is no decision procedure which can be used in the general case to determine whether a condition containing variables is always *false* regardless of how the variables are instantiated.

5.2.2 The Semantic Approach

A second method for determining the requirements of a particular component c from a slice expression S is the semantic approach:

1. Construct the transition system model $\mathfrak{M}(S)$
2. From $\mathfrak{M}(S)$ construct a model \mathfrak{M}_c which represents the behaviour of component c .
3. From \mathfrak{M}_c construct a specification B_c which represents the behaviour of component c .

Given that the original slice S is written in LOTOS, the operational semantics of LOTOS can be applied to construct the transition system model $\mathfrak{M}(S)$. (Various tools are available for automating this process [62,82].) Simply by executing the slice S and observing how the component c interacts with the other components the designer gains insight into the behaviour of c .

Generating the *traces* of component c can easily be done by generating the *traces* of $\mathfrak{M}(S)$ and then projecting them through the interactions of c . This corresponds to taking a timeline and projecting the timeline onto a single axis (figure 5.13). Given that B_c is the subexpression of the resource oriented specification $B_{\langle IP, C \rangle}$ which represents the behaviour of component c , this projection must be a valid trace of the B_c ; defining $I(c)$ as the set of interactions in which component c is involved this can be stated as the following theorem.

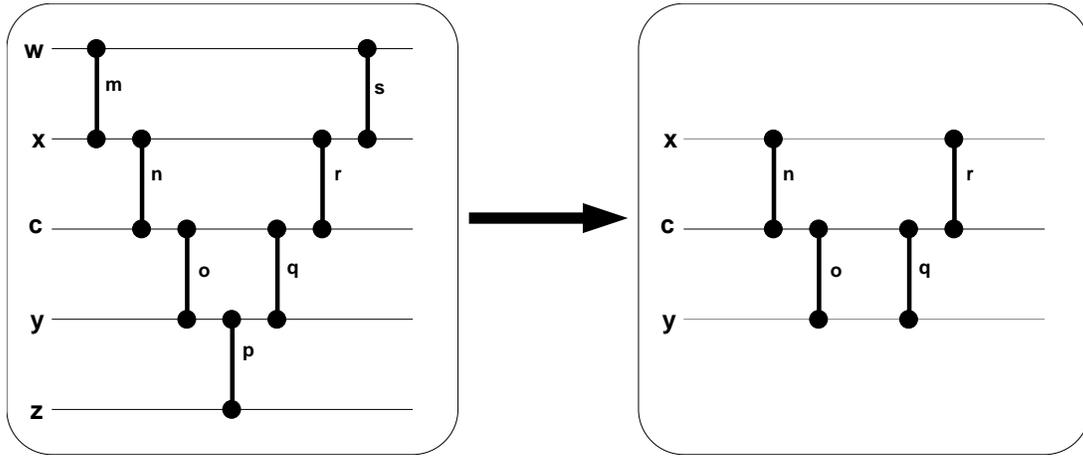


Figure 5.13 Projecting a timeline trace onto interactions of component c .

Theorem: 2

$$\forall t \in \text{traces}(S) \bullet t \upharpoonright I(c) \in \text{traces}(B_c)$$

However, as shown in §4.2.1.1 and §5.1.1.2, *traces* do not capture all aspects of a component's behaviour. A model based on *failures* is more accurate. The problem then becomes looking at the set $\text{failures}(S)$ and then trying to determine what this set implies about $\text{failures}(B_c)$.

To determine whether the pair $\langle t_c, A \rangle$ can be a failure of B_c while still maintaining the requirement that $B_{\langle IP, C \rangle} \text{ext } S$, it is necessary to look at the set $T(t_c)$ of all traces t of S which contain t_c , i.e.,

$$T(t_c) = \{t \in \text{traces}(S) \mid t \upharpoonright I(c) = t_c\}$$

Then for every t in $T(t_c)$, $\langle t_c, A \rangle$ can be a *failure* of B_c only if $\langle t, A \rangle$ is a *failure* of S . This can be restated as the following theorem.

Theorem: 3

For all components $c, t \in \text{traces}(S)$, $A_c \subseteq I(c)$
 if $\langle t \upharpoonright I(c), A_c \rangle \in \text{failures}(B_c)$
 then $\forall u \in \{v \in \text{traces}(S) \mid v \upharpoonright I(c) = t \upharpoonright I(c)\} \bullet \langle u, A_c \rangle \in \text{failures}(S)$

Given a slice specification S an algorithm for generating *failures* for B_c can be given as follows. The input to the algorithm is a trace t_c of the component; the output is a set of possible *refusals* of B_c after executing trace t_c . The algorithm is illustrated using the example of figure 5.14 where a design structure of three components, and a slice expression represented as a transition system are given; the set of *refusals* is computed in the example assuming that $t_c=m$.

1. Generate the set $T(t_c)$ which are the traces of S which project onto t_c , i.e., $T(t_c) = \{t \in \text{traces}(S) \mid t \upharpoonright I(c) = t_c\}$. In the example, if $t_c=m$ then $T(m)=\{a.m, b.m\}$
2. Generate F a subset of $\text{failures}(S)$ such that the traces in F are elements of $T(t_c)$ i.e.,

$$\text{given } t_c, \text{ let } F = \{\langle t, A_c \rangle \in \text{failures}(S) \mid t \in T(t_c)\}$$

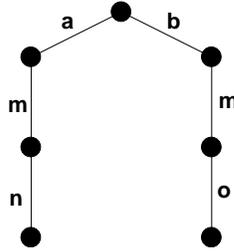
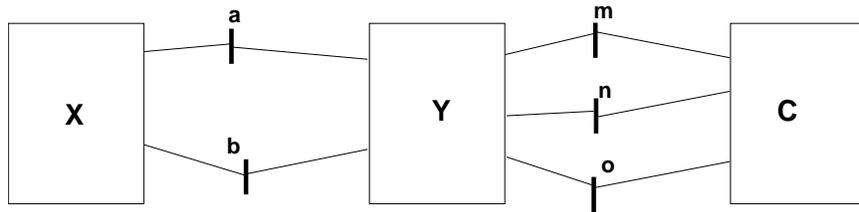
In the example, the set F is then defined as follows:

$$F = \{\langle a.m, \emptyset \rangle, \langle a.m, \{o\} \rangle, \langle a.m, \{m\} \rangle, \langle a.m, \{m, o\} \rangle, \\ \langle b.m, \emptyset \rangle, \langle b.m, \{n\} \rangle, \langle b.m, \{m\} \rangle, \langle b.m, \{m, n\} \rangle\}$$

3. Partition F into those *failures* which have the same *traces*, i.e., $\langle t_1, A_1 \rangle$ and $\langle t_2, A_2 \rangle$ are in the same partition if and only if $t_1 = t_2$. Partitioning F in the example gives the two partitions P_1 and P_2 :

$$P_1 = \{\langle a.m, \emptyset \rangle, \langle a.m, \{o\} \rangle, \langle a.m, \{m\} \rangle, \langle a.m, \{m, o\} \rangle\}$$

$$P_2 = \{\langle b.m, \emptyset \rangle, \langle b.m, \{n\} \rangle, \langle b.m, \{m\} \rangle, \langle b.m, \{m, n\} \rangle\}$$



Given that set $t_c=m$

1. Generate set $T(m)$

$$T(m) = \{a.m, b.m\}$$

2. Generate set F

$$F = \{\langle a.m, \emptyset \rangle, \langle a.m, \{o\} \rangle, \langle a.m, \{m\} \rangle, \langle a.m, \{m, o\} \rangle, \\ \langle b.m, \emptyset \rangle, \langle b.m, \{n\} \rangle, \langle b.m, \{m\} \rangle, \langle b.m, \{m, n\} \rangle\}$$

3. Partition set F

$$P_1 = \{\langle a.m, \emptyset \rangle, \langle a.m, \{o\} \rangle, \langle a.m, \{m\} \rangle, \langle a.m, \{m, o\} \rangle\}$$

$$P_2 = \{\langle b.m, \emptyset \rangle, \langle b.m, \{n\} \rangle, \langle b.m, \{m\} \rangle, \langle b.m, \{m, n\} \rangle\}$$

4. Compare partitions to determine $failures(B_c)$

$$\{\langle m, \emptyset \rangle, \langle m, \{m\} \rangle\}$$

Figure 5.14 Generating the $failures(S_c)$ from $failures(S)$

4. The allowable elements of $failures(B_c)$ implied by S are the elements of the partitions which have the same refusal set, i.e., given that F is partitioned into n sets P_1 to P_n , the allowable failures of c with trace t_c is given by the set $\{\langle t_c, A \rangle \mid \forall k \in \{1, \dots, n\} \bullet \exists t \bullet \langle t, A \rangle \in P_k\}$. In the example, since $\langle a.m, \emptyset \rangle \in P_1$ and $\langle b.m, \emptyset \rangle \in P_2$ therefore $\langle m, \emptyset \rangle$ is a possible *failure* of component c . Similarly, since $\langle a.m, m \rangle \in P_1$ and $\langle b.m, m \rangle \in P_2$ therefore $\langle m, m \rangle$ is also a possible *failure* of component c .

5.2.3 Discussion

Given a slice expression S , this section has explored two approaches for generating the component specification S_c for component c from S : the syntactic approach; and the semantic approach. Although both of these approaches are applicable to the problem of deriving component behaviour, the following observations can be made when comparing the two methods:

- The syntactic transformation ϕ_c can be applied to any LOTOS specification regardless of the size of the corresponding model. Also, the transform is defined in such a way that it is efficient to apply to a LOTOS specification and can be partially automated. The semantic approach to deriving the behaviour S_c requires the model of S to be generated. In many cases this model will be large or even of infinite size and is thus not practical to generate.
- Generating the behaviour of component c from the semantic model of S required a rather complex manipulation of the model $\mathfrak{M}(S)$, where sets of *traces* and *failures* were generated, compared and manipulated. Using the syntactic approach, generating the behaviour of component c can be done simply by executing the specification $\phi_c(S)$.
- Specifications written in a language such as LOTOS are generally structured so as to be understandable to a reader, through appropriate use of the different specification styles (see §2.3.1). In the syntactic approach, the structure of the specification S is maintained in the derived specification $\phi_c(S)$. Thus,

by comparing $\phi_c(S)$ to S a designer can determine from where a particular behaviour pattern of $\phi_c(S)$ has been derived. In the semantic approach, the structure of S is lost when the transition system $\mathfrak{M}(S)$ is created. Thus given a particular behaviour pattern of c derived from $\mathfrak{M}(S)$, the designer cannot trace back to S the construct which led to that behaviour pattern of c .

- In the syntactic approach, $\phi_c(S)$ is a LOTOS specification which can be used as a basis for creating the complete behaviour specification of component c . In the semantic approach, the result is a set of *failures* for component c ; from this set the designer must work backwards to generate the LOTOS specification for the component.
- The syntactic transform $\phi_c(S)$ does not always correctly reflect all the behavioural requirements of component c implied by S . In particular: the *traces* of $\phi_c(S)$ may contain more *traces* than that implied by S ; and certain operators such as choice, may cause $\phi_c(S)$ to introduce nondeterminism into the behaviour of component c where none was implied by S (see §5.2.1.4). The semantic approach will more accurately derive the behavioural requirements of the component from the slice S .

5.2.3.1 Nondeterminism

One problem that arose in trying to extract the component behaviour from a slice expression was how to handle the issue of nondeterminism. The problem is that when nondeterminism is encountered in a slice S , how is the nondeterminism assigned to individual components.

A system is nondeterministic if there is a trace t and an interaction e such that after executing trace t interaction e may be offered or it may be refused; both possibilities are correct and which option is chosen is left as a nondeterministic choice of the system. Formally nondeterminism can be defined as follows.

Definition: nondeterminism.

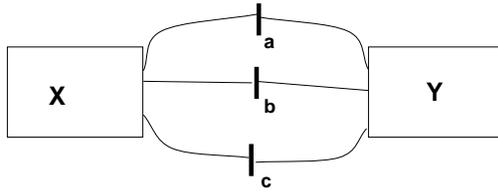
A transition system B is *nondeterministic* if and only if there exists a trace t and an event e such that $\langle t, e \rangle \in failures(B)$ and $t.e \in traces(B)$. A system which is not nondeterministic is *deterministic*.

If a slice expression S is nondeterministic what does this imply regarding the behaviour of individual components? Does this imply that all components are nondeterministic? In general when there is nondeterminism within a slice expression this will translate into nondeterminism within a single component of the system; the other components will remain deterministic. Thus for each point of nondeterminism within a slice expression a decision must be made as to which component will resolve the nondeterminism. A slice expression in general does not contain the information required to make such a decision automatically.

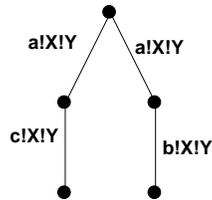
The problem is illustrated by the example of figure 5.15. The structure consists of two components connected by three IPs. The first behaviour expression is a slice expression for the system. This slice expression is nondeterministic since after executing interaction $a!x!y$ the interaction b may be offered and c refused, or vice versa. If we now apply the transforms ϕ_X and ϕ_Y (and noting that $\phi_X(S) = \phi_Y(S) = S$) the nondeterminism of the original slice is contained within both component x and component y . The composition of these expressions (shown as the bottom behaviour expression in figure 5.15) is *not* an extension of S . The resource oriented expression:

$$(\phi_X(S) ||| default(X)) || (\phi_Y(S) ||| default(Y))$$

has as one of its failures $\langle a, \{b, c\} \rangle$ but this is not a failure of S . The problem is that the transformation should determine that there is nondeterminism within the expression S and then make exactly one of the components X or Y nondeterministic; the other should be deterministic. Which component should be nondeterministic is not indicated within S and requires a decision to be made by the designer.

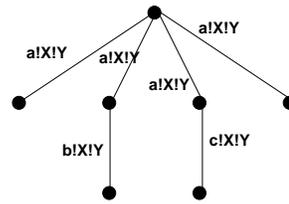


(* Slice containing nondeterminism *)
 a!X!Y; c!X!Y; stop
 []
 a!X!Y; b!X!Y; stop



(* composition of component behaviours
 derived from slice expression *)

(*behaviour of component X*)
 (((a!X!Y; b!X!Y; stop)
 []
 (a!X!Y; c!X!Y; stop)) ||| default[...](X))



||

(*behaviour of component Y*)
 (((a!X!Y; b!X!Y; stop)
 []
 (a!X!Y; c!X!Y; stop)) ||| default[...](Y))

Figure 5.15 Nondeterministic slice expression.

5.3 Races and Critical Races

One problem often encountered in concurrent systems is that of *critical races* [39,54,53]. A race occurs if a particular sequence of input events to a system can result in different output sequences from the system depending on the speed with which different concurrent components execute. A race is *critical* if the designer has assumed a specific output sequence of events given the particular input sequence.

Comparing slice expressions to resource oriented behaviour expressions can assist a designer in detecting critical races in a design. Slice expressions define the behaviour of a system in terms of sequences of events which the designer expects to occur between sets of components. When the concurrency of components is taken into account, it is possible to discover new interaction sequences in which the system may engage.

The problem of races is illustrated in figure 5.16. The top illustration shows two slice expressions of the producer/consumer example operating in parallel. One slice shows value x being passed from components `prod1` to `buff` and then to `cons1`; the other slice is a similar behaviour involving value y being passed from components `prod2` to `buff` to `cons2`. Because all components are concurrent, a system which provides the slice behaviour of the top figure may also allow the behaviour illustrated at the bottom of the figure where the values x and y ‘cross over’ and end up at different destinations. If the designer made the assumption that values x and y always arrive at `cons1` and `cons2` respectively, then reversing the destinations is a race which is critical.

Given a slice expression S and a structural description $\langle IP, C \rangle$ the method for detecting races is the following:

1. For each $c \in C$ construct the component expression $\phi_c(S)$. These component expressions represent the ‘minimum behaviour’ which component c must exhibit in order to satisfy the behaviour implied by slice S .

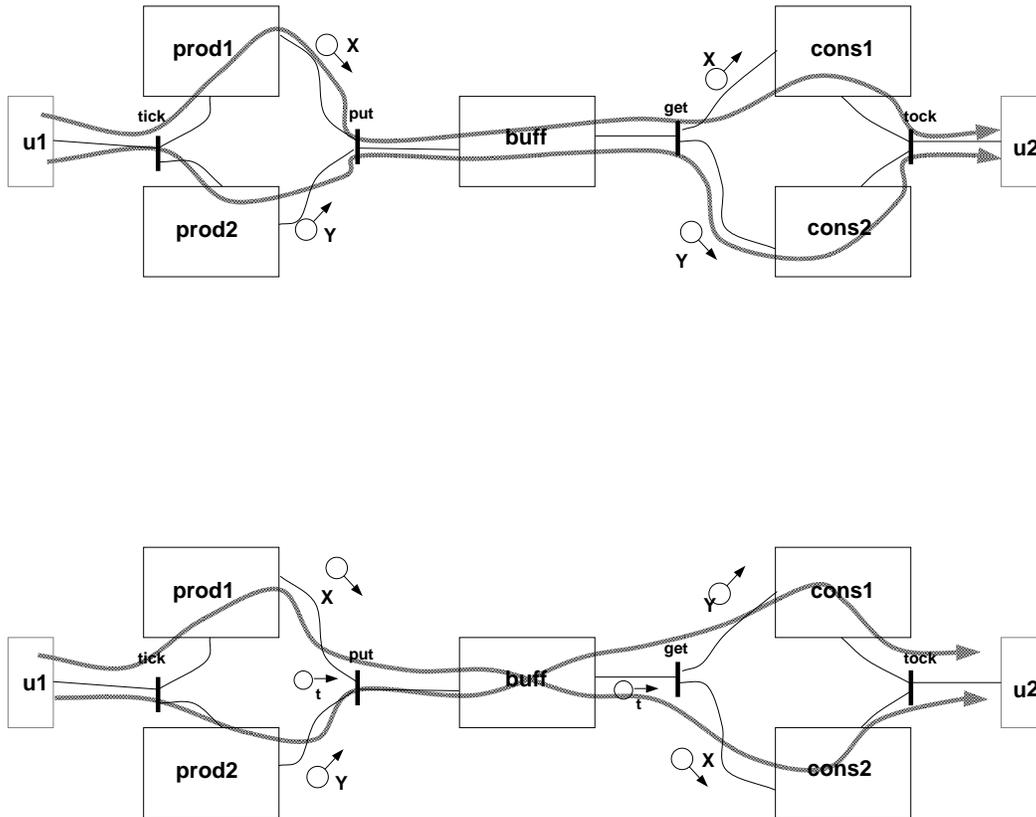


Figure 5.16 Illustration of race condition.

2. Compose the component expressions $\phi_c(S)$ into a resource oriented expression of the system behaviour. This expression is defined as the *component closure* of S relative to $\langle IP, C \rangle$ and will be denoted as $\phi(S)$.
3. Detect race conditions by comparing $\phi(S)$ with S ; these race conditions can then be presented to the designer to determine whether they are critical races.

5.3.1 Constructing the Component Closure

Given a slice expression S and a structural description $\langle IP, C \rangle$ what constitutes the ‘minimum behaviour’ which must be exhibited by $\langle IP, C \rangle$ in order that the resource oriented behaviour of the system is an *extension* of S ? The steps for constructing the component closure are illustrated in figure 5.17 and are defined as follows:

1. For each $c \in C$ construct the behaviour expression $\phi_c(S)$. This behaviour expression represents the behaviour of c implied by S .
2. Compose all the expressions $\phi_c(S)$, $c \in C$ into a resource oriented expression representing the behaviour of $\langle IP, C \rangle$. This expression, denoted as $\phi(S)$, is called the component closure.

Definition: Component Closure.

Given a slice expression S and a structure $\langle IP, C \rangle$, the expression

$$\phi(S) = \parallel_{c \in C} (\phi_c(S) \parallel \text{default}(c))$$

is defined as the *component closure* of S relative to $\langle IP, C \rangle$.

The significance of the component closure $\phi(S)$ is that it defines a minimum behaviour that a system must exhibit in order to provide the behaviour of S . The meaning of ‘minimum behaviour’ is that any resource oriented specification $B_{\langle IP, C \rangle}$ must be an *extension* of $\phi(S)$. This is shown by the following theorem.

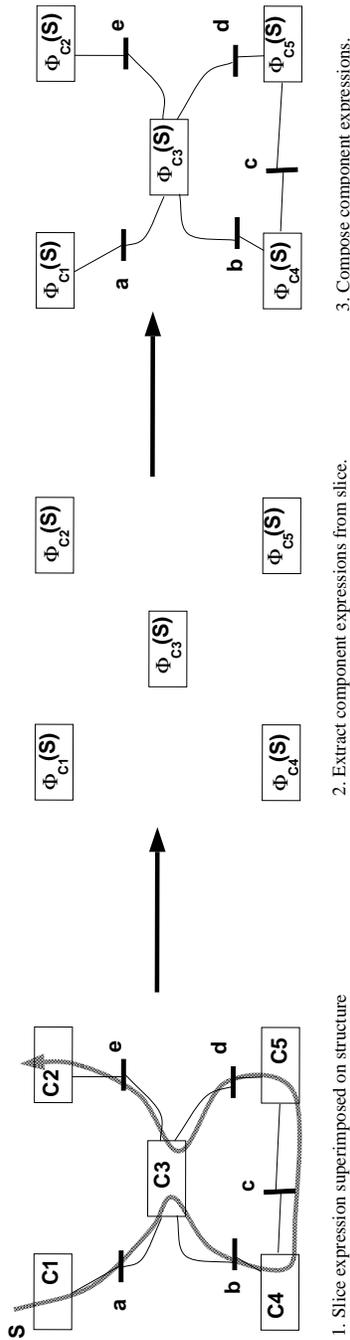


Figure 5.17 Constructing the component closure.

Theorem: 4

$$B_{\langle IP, C \rangle} \text{ ext } \phi(S).$$
5.3.2 Race Conditions

The *traces* of a slice expression S is a subset of the *traces* of the component closure $\phi(S)$ as shown by the following theorem.

Theorem: 5

$$\text{traces}(\phi(S)) \supseteq \text{traces}(S).$$

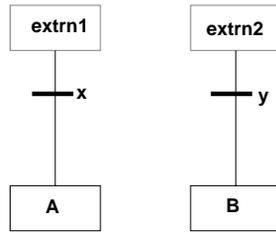
Clearly it is possible for a slice expression S to exist where the relationship of theorem 5 is a proper subset, i.e., there are traces of the component closure which are not traces of the slice expression. This happens due to *race conditions* with new traces introduced by the concurrency implied by the structure of the components and the data flow direction and anonymous components introduced by the designer.

A simple example of a race condition is illustrated by the structure of figure 5.18. The design structure consists of two components **A** and **B** connected to the external environment by IPs **x** and **y** respectively. The timeline and corresponding LOTOS behaviour expression show **A** interacting at **x** followed by **B** interacting at **y**. The component closure is defined as:

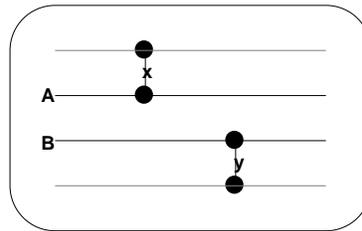
```

specification componentClosureS[x,y]
  behaviour
    { * behaviour of component A * }
    ((x !A ?C1:component; exit) ||| default[x](A))
    ||
    { * behaviour of component B * }
    ((y !B ?C2:component; exit) ||| default[y](B))

```



a) Design Structure.



b) Timeline illustrating design behaviour.

```

x !A ?C1:component;
y !B ?C2:component;
exit

```

c) LOTOS expression corresponding to timeline.

Figure 5.18 Structure exhibiting race condition.

Comparing the component closure and the slice specification it can be seen that the sequence:

```
y !B !Extrn2;  
x !A !Extrn1
```

is a trace of `componentClosures` but is not a trace of `s`. This is a rather obvious (and uninteresting) conclusion and simply states that the interactions at `y` and `x` are independent of each other and regardless of the fact that the designer drew the timeline showing the interactions in a particular order, they can occur in either order.

Formally, a *race* can be defined as follows.

Definition: Race.

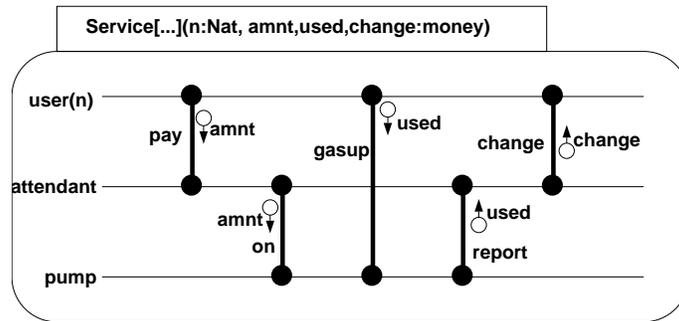
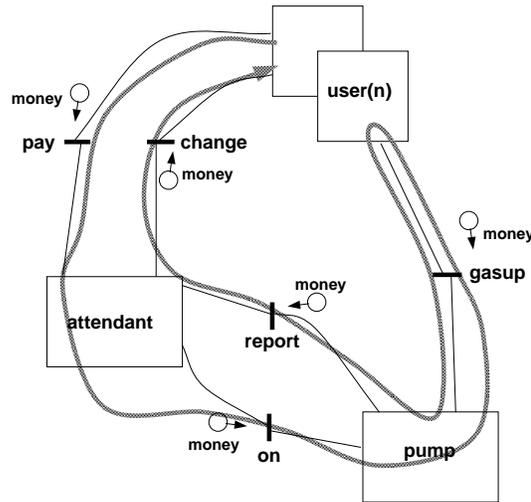
Trace t is a *race* for slice expression S if and only if

$$t \in \text{traces}(\phi(S)) \text{ and } t \notin \text{traces}(S).$$

A simple race condition is unlikely to be of great significance to the designer, i.e., it is unlikely to be critical. It generally means that interactions of unrelated components can occur in any order due to the modeling of concurrency by interleaving. A race is not necessarily an error condition nor a condition which the designer need be concerned about.

There are conditions however where a race condition is far more serious and indicative of a possible critical race. A simple example is the gas station problem [39] as illustrated by the structural design in figure 5.19. The problem is to design a self-serve gas station where the customer wishing to fill his tank prepays an attendant. The attendant then turns on the gas pump allowing the customer to pump gas. After pumping gas, the customer returns to the attendant to get a receipt and to receive any change due.

A typical scenario is illustrated by the timeline of figure 5.19. A customer `user(n)` prepays an `attendant` at `IP pay`. The attendant then turns the `pump` on (`IP`



```

process Service[...] (n:Nat, amnt,used,change:money) : exit :=
  pay !usr(n) !attendant !amnt;
  on !attendant !pump !amnt;
  gasup !pump !usr(n) !used;
  report !pump !attendant !used;
  change !attendant !usr(n) !change;
  exit

```

Figure 5.19 Critical race example.

```

service[...] (1,20,10,10)
  |||
service[...] (2,10,5,5)

where
  process service[...] (n:Nat, amnt,used,change:money) : exit :=
    .
    .
    .

```

Figure 5.20 Slice expression showing two customers buying gas.

on). The customer pumps gas (IP `gasup`). The `pump` then reports to the `attendant` the actual amount of gas pumped (IP `report`), and the attendant then returns any change required to the customer (IP `change`). The LOTOS process corresponding to this timeline is shown underneath the timeline.

A designer can now create a more complex slice expression from the timeline by having two customers requesting gas at the same time while paying different amounts of money and pumping different amounts of gas (figure 5.20). The component closure of this slice expression can be generated (figure 5.21). The component closure takes into account data flow. It also makes the `user` components anonymous to the `attendant` and `pump`; thus the `attendant` and `pump` will provide a service to any requesting `user` without restricting with which `user` they interact. (This is the assumption which will actually cause the critical race; the attendant will no longer be able to specify which user receives the change from a particular transaction.)

Analysis of the component closure and the slice expression reveals that the trace of figure 5.22 is a trace of the slice expression but is not a trace of the component closure. Moreover, this trace *is* indicative of a design error caused by races between the concurrent components. The error occurs because there is no way of guaranteeing that a particular `user` receives the correct change, e.g., in the above example `user(1)` has received the change due `user(2)`.

```

(user[...] (1,20) ||| default[...] (user(1)))
||
(user[...] (2,10) ||| default[...] (user(1)))
||
(attendant[...] (10,5) ||| default[...] (attendant))
||
(pump[...] (10,5) ||| default[...] (pump))

where
  process user[...] (n:Nat, amnt,pumps:money) : exit :=
    service_user[...] (n,amnt,pumps)

  where
    process service_user[...] (n:Nat,amnt,pumps:money) : exit :=
      pay !usr(n) ?c1:Component !amnt;
      gasup ?c2:Component !usr(n) !pumps;
      change ?c3:Component !usr(n) ?c4:money;
      exit
    endproc
  endproc

  process attendant[...] (change1,change2:money) : exit :=
    service_attendant[...] (change1)
    |||
    service_attendant[...] (change2)

  where
    process service_attendant(change:money) : exit :=
      pay ?c1:Component !attendant ?c2:money;
      on !attendant ?c3:component !c2;
      report ?c4:component !attendant ?c5:money;
      change !attendant ?c6:Component !change;
      exit
    endproc
  endproc

  process pump[...] (used1,used2:money) : exit :=
    service_pump[...] (used1)
    |||
    service_pump[...] (used2)
  where
    process service_pump(used:money) : exit :=
      on ?c1:Component !pump ?c2:money;
      gasup !pump ?c3:Component;
      report !pump ?c4:Component !used;
      exit
    endproc
  endproc

```

Figure 5.21 Component closure for gas station example.

```

pay !user(1) !attendant !20;
on !attendant !pump !20;
  gasup !pump !user(1);
  report !pump !attendant !10;
  pay !user(2) !attendant !10;
  on !attendant !pump !10;
    gasup !pump !user(1);
    report !pump !attendant !5;
    change !attendant !user(1) !5

```

Figure 5.22 Trace exhibiting a critical race.

The trace of figure 5.22 is an example of a *serious race*. A serious race occurs if a trace of component c generated by the component closure $\phi(S)$ is not contained within any traces of S . Thus there is a behaviour pattern for component c within the component closure and this behaviour of c cannot be seen as part of S . In the example trace of 5.22 a serious race can be detected by looking at the behaviour of `user(1)`. The trace of figure 5.22 when restricted to interactions involving `user(1)` gives the trace:

```

pay !user(1) !attendant !20;
  gasup !pump !user(1) !10;
  change !attendant !user(1) !5

```

However this trace is not contained within any trace of the corresponding slice expression.

This leads to the following formal definition of a serious race.

Definition: Serious race.

If S is a slice expression, and $t \in \text{traces}(\phi(S))$ then t is a *serious race* of S if and only if there exists a $c \in C$ such that for all $u \in \text{traces}(S)$, $u \upharpoonright I(c) \neq t \upharpoonright I(c)$. A slice expression S is *serious race free* if it does not contain any serious races.

A serious race may or may not be critical. An analysis of the component closure can detect serious races; a designer can then determine whether a particular serious race is critical.

5.4 Verifying Component Specifications

A typical design process for concurrent system involves first generating a set of slice expressions showing how the components interact with each other, and then constructing the component behaviour expressions which define the behaviour of each component of the structure. Given a slice expressions S , and the individual component specifications B_c , how can S be used to verify that each of the component specifications is correct?

By assumption 2 (page 116), $B_{\langle IP, C \rangle}$, which is the composition of the component expressions, must be an *extension* of S . Therefore one method of verifying the component expressions is to compose them into $B_{\langle IP, C \rangle}$ and verify that this expression *extends* S . One problem with such a verification is that it requires all the component specifications to be written before any verification occurs. Normally, the component specifications are completed at different times and possibly written by different design teams. It is therefore desirable to be able to verify the component specifications as they are written in order to discover design errors in components as early as possible in the system design life cycle.

Given a slice expression S and a component behaviour specification B_c for component c there are two ways of verifying that B_c does not violate the constraints imposed by S :

1. Using assumption 3 (page 126), show that $B_c \text{ ext } \phi_c(S)$. If B_c does not *extend* $\phi_c(S)$ then B_c is incorrect.
2. Using theorem 3 (page 140), compare the sets $failures(B_c)$ with $failures(S)$; if there are elements of $failures(B_c)$ which violate theorem 3 then B_c contains an error.

Using the first verification technique requires a method for verifying the *ext* relation between behaviour expressions. Practical aspects of this verification are discussed further in Chapter 7.

When using theorem 3 to verify component specification B_c , the theorem can be restated as follows:

if

$$\langle t_c, A_c \rangle \in failures(B_c) \text{ and}$$

$$\exists t \in \{u \in traces(S) \mid [u \upharpoonright I(c)] = t_c\} \bullet \langle t, A_c \rangle \notin failures(S)$$

then *not* $B_{\langle IP, C \rangle} \text{ ext } S$.

By comparing the failures of B_c with the failures of S it is possible to determine whether the implicant of the above statement is true. If the implicant is true then $B_{\langle IP, C \rangle}$ cannot be an extension of S and therefore there must be an error in the behavioural specification B_c . The process for performing such a verification is the following:

- Generate a *failure* $\langle t_c, A_c \rangle$ of B_c .
- Search for a trace t of S such that $t_c = t \upharpoonright I(c)$.
- If $\langle t, A_c \rangle$ is not a failure of S then the specification B_c is in error.

Unfortunately, even if all the component specifications satisfy the two verification techniques of the previous page, this is not a sufficient condition for $B_{\langle IP, C \rangle} \text{ ext } S$. The reason for this is that if the slice expression S contains nondeterminism, the intention of the designer is that the nondeterminism should be included in only one component of the interaction (see §5.2.3). If the nondeterminism is included in more than one component of the interaction then all components may satisfy the two verification conditions individually, but it will not be true that $B_{\langle IP, C \rangle} \text{ ext } S$. This is what occurred in the example of figure 5.15. In order to detect such problems with nondeterminism it is necessary to compose

the component specifications and then verify the composition against the slice expression. Since component specifications are created at different times it is desirable to have a method of composing a partial set of component specifications, and then verifying the composition of this partial set.

Assume we have a structural specification $\langle IP, C \rangle$. The component set C can be partitioned into those components for which component behavioural specifications have been developed, and those components for which the component behavioural specifications are not yet developed. Let $D \subseteq C$ represents the set of components for which complete component behavioural specifications have been developed. The expression

$$\parallel_{c \in D} (B_c \parallel \text{default}(c))$$

is the composition of the currently developed component specifications. (See figure 5.23 for an example).

Given a slice expression S and a set of component expressions $B_c, c \in D$ what is the relation between the slice expression S and the expression

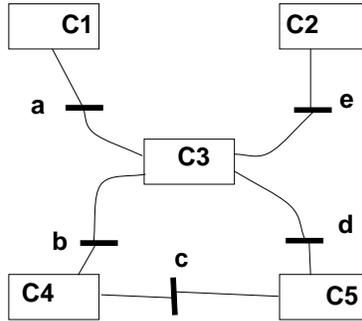
$$B_D = \left(\parallel_{c \in D} (B_c \parallel \text{default}(c)) \right)$$

Let $I(D)$ be the set of interactions involving only the components of D , i.e., $I(D) = \{e \in I \mid \forall c \notin D \bullet e \notin I(c)\}$. We can now prove the following theorem.

Theorem: 6

For all $D \subseteq C$, $t \in \text{traces}(S)$ and sets of interactions A_D involving only components of D
 if $\langle t \upharpoonright I(D), A_D \rangle \in \text{failures}(B_{\langle IP, D \rangle})$
 then $\langle t, A_D \rangle \in \text{failures}(S)$

Theorem 6 provides a basis for verifying a set of component expressions $B_c, c \in D$ against a slice expression. The theorem can be stated as follows:



If $D = \{c_2, c_3, c_4\}$ then the behaviour expression B_D is defined as:

```

(Bc2 | | | default [...] (c2))
  | |
(Bc3 | | | default [...] (c3))
  | |
(Bc4 | | | default [...] (c4))

```

Figure 5.23 Composing component specifications for a subset of the components.

if
there exists $D \subseteq C$, $t \in \text{traces}(S)$, $A_D \subseteq I(D)$ such that
 $\langle (t \upharpoonright I(D)), A_D \rangle \in \text{failures}(B_D)$ and
 $\langle t, A_D \rangle \notin \text{failures}(S)$
then
 $B_{\langle IP, C \rangle}$ is not an *extension* of S .

The testing of the components B_c , $c \in D$ then involves the following steps:

1. Generate a *failure* $\langle t_D, A_D \rangle$ of B_D .
2. Search for a trace t of S such that $t_D = t \upharpoonright I(D)$.
3. If $\langle t, A_D \rangle$ is not a failure of S then one of the specification B_c is in error.

5.5 Chapter Summary

This chapter has developed some theoretical results which can be used as a basis for integrating slices into a process for the design of concurrent systems. The most important results of this chapter show how slice expressions can be used for the following purposes:

- **Developing individual component specifications.** Component specifications can be derived from a slice expression S in two ways: syntactically or semantically. In the syntactic approach, a transform $\phi_c(S)$ is defined such that $\phi_c(S)$ represents the behaviour of component c implied by S . In the semantic approach, the set of $\text{failures}(S)$ is generated, and from this a possible set of *failures* of component c is derived.
- **Analyzing a design for critical races.** Given a slice expression S and a design structure $\langle IP, C \rangle$, this chapter presented a method for detecting potential serious races in the design; these serious races can then be presented to the designer to determine whether they are critical. The method consists of:
 1. For each $c \in C$ derive from S the component specifications $\phi_c(S)$.
 2. Compose the component specifications into the component closure $\phi(S)$.

3. Compare the traces of each component within $\phi(S)$ to the traces of each component within S . If more traces of component c are found in $\phi(S)$ than in S , then a potential critical race exists.
- **Verifying component specifications using slices.** Each slice expression S imposes conditions on the behaviour of the components of a system. Since these component specifications are developed at different times, it is desirable to verify the component specification B_c as soon as it is developed rather than waiting until all the component specifications have been developed. Three ways were given for verifying component specification B_c relative to slice S :
 1. Verify that $B_c \text{ ext } \phi_c(S)$.
 2. Generate the set $failures(B_c)$ and compare this to the set $failures(S)$.
 3. Letting D be the set of components for which specifications have been completed, and the behaviour expression B_D be the composition of these completed component specifications, compare the set $failures(B_D)$ to the set $failures(S)$.

Chapter 6: Application of the Slice Techniques: An Example

This chapter illustrates by example how the formal techniques developed in chapter 5 can be applied to a slice oriented design process.

The design process of interest to this research begins when the designer understands the requirements which the system is to satisfy. The design process ends when the designer has constructed the structure hierarchy tree down to the level of the primitive components, specified the interconnection between the components, and has specified the behaviour of each of these primitive components

The steps which a designer uses to progress from the requirements to the design are identified in 3.1 as the following:

1. Specify the structural design of the system.
2. Explore design behaviour by developing slice expressions showing component interactions.
3. Generate component behaviour expressions.
4. Verify the component specifications using the slice expressions.
5. Decompose components as required.

To illustrate these concepts the design of an (overly simplified) telephone system is used as an example. (This example is a simplified version of the telephony case study described in [26] and summarized in Appendix D.) The requirements of interest in the telephony system are the following:

- There are a number of phones which can be used for placing and receiving calls. Each phone has a unique identifier.
- Each phone has a handset which can be lifted off or replaced on the hook.
- A user can request that a call be placed to another phone. The system will record that a call has been requested and either service the request, or notify

the requesting phone that the request could not be completed. A call cannot be completed if the other phone is busy, i.e., the handset is off the hook.

- A phone will never be engaged in more than one call at a time.

6.1 Specifying the Structural Design

Once the requirements of a system have been analyzed the next step in the design process is the development of the structural design. A system's structural design specifies the following aspects of a system:

- The components from which a system is constructed.
- The interconnection mechanisms between the components.
- The interconnection topology.

The steps involved in a structural design can be broken down as follows:

1. Identify the components.
2. Identify IPs by which components communicate.
3. Wire the components together by connecting the components to the IPs.
4. Determine the characteristics of the wiring, in particular the data parameters passed between components at an IP during an interaction.

For the telephony example a typical component decomposition dictated by the distributed nature of the system is to have one component for each phone in the system and one component for the central switch. Once this decision is made it can be recorded on the structural diagram (figure 6.1). The figure shows a system consisting of three phone components (`phone(n)`) and a `switch`. Three external components (`user(n)`) are used to model users of the system.

Once the system components have been identified the designer can specify the interaction points between components. Identification of the interaction points requires a designer to determine which components must interact directly

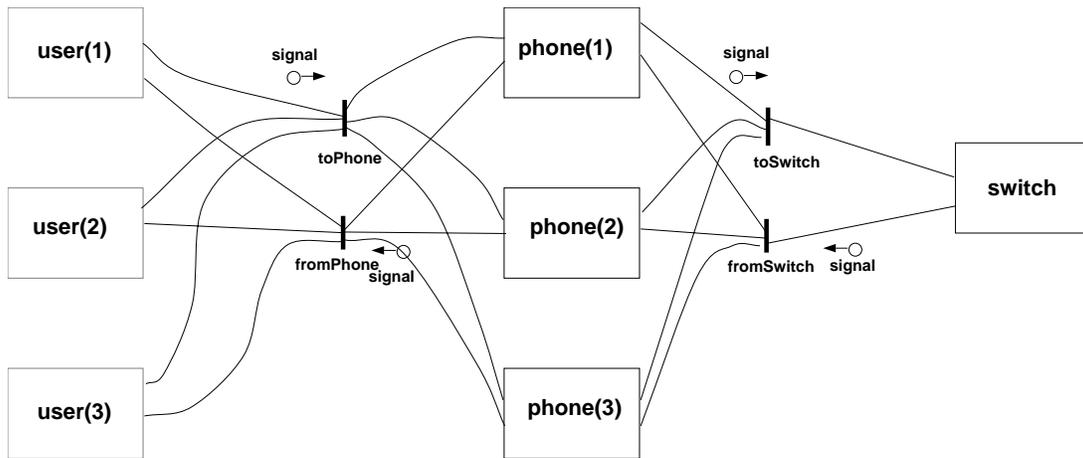
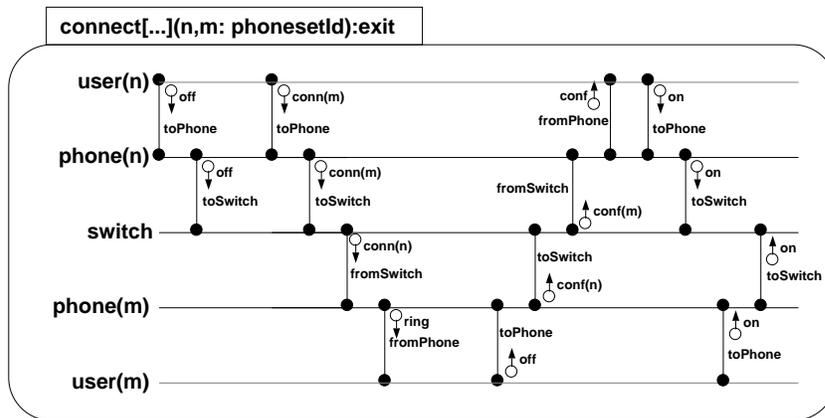
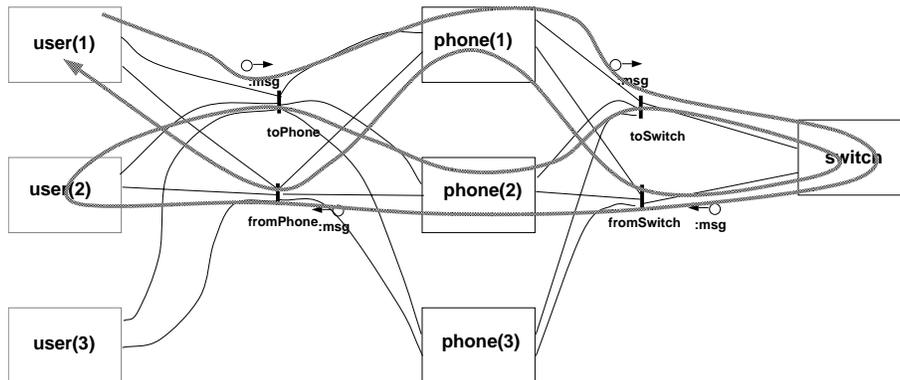


Figure 6.1 Structural design of a simple telephone system.

with each other and what data must be passed between components during the interaction. The structure diagram of figure 6.1 shows four IPs:

- **toPhone**: **user** sends signals to **phone**.
- **fromPhone**: **phone** sends signals to **user**.
- **toSwitch**: **phone** sends signals to **switch**.
- **fromSwitch**: **switch** sends signals to **phone**.

When identifying IPs it is often useful to draw some timelines to explore how the components must interact with each other in order to provide the necessary system services. The purpose of these timelines is to investigate with simple sequential behaviour how events ripple through the components of the system, and what IPs and data flow will be required. Figure 6.2 is a timeline showing a successful call connection from **user(1)** to **user(2)**. The top illustration shows the structural design with a slice representing a call connect superimposed on the structure; below this is a timeline which defines a LOTOS process `connect(n,m)` showing the interaction sequences required to make the call; at the bottom of the figure is the LOTOS behaviour expression `connect` defined using this process. Note that the definition of the process `connect(n,m)` can be generated directly from the timeline.



```

specification connect[...] :=
behaviour
  connect [toPhone, fromPhone, toSwitch, fromSwitch] (1, 2)
  where
    process connect [toPhone, fromPhone, toSwitch, fromSwitch]
      (n,m:phonesetid) : exit :=
      toPhone !user(n) !phone(n) !off;
      toSwitch !phoneset(n) !switch !off;
      toPhone !user(n) !phone(n) !conn(m);
      toSwitch !phone(n) !switch !conn(m);
      fromSwitch !switch !phone(m) !conn(n);
      fromPhone !phone(m) !user(m) !ring;
      toPhone !user(m) !phone(m) !off;
      toSwitch !phone(m) !switch !conf(n);
      fromSwitch !switch !phone(n) !conf(m);
      fromPhone !phone(n) !user(n) !conf;
      toPhone !user(n) !phone(n) !on;
      toSwitch !phoneset(n) !switch !on;
      toPhone !user(m) !phone(m) !on;
      toSwitch !phoneset(m) !switch !on;
      exit
  endproc

```

Figure 6.2 Slice showing simple connection establishment: telephony example.

6.2 Exploring Design Through Slices

Slices provide a means for a designer to explore the implications of structural and behavioural design decisions without necessarily having to generate complete specifications of individual components. As described in §3.1.2, behavioural design exploration can begin by generating simple slices illustrating the basic functionality of the system, and then refining these slices in order to specify more complex behaviour patterns. The two methods for refining slices to express complex behaviour patterns were identified in §3.1.2 as:

- *Slice extension.* Beginning with slice expression S_1 , a designer constructs slice S_2 which adds new behaviour to the slice S_1 , but does not violate any behaviours specified by S_1 .
- *Slice combination.* Beginning with slice expressions S_1 and S_2 , a designer combines these slices into slice $S_1\#S_2$ which defines how the slices S_1 and S_2 interact with each other when they are executing concurrently.

These refinement methods will be illustrated by developing a number of different slice expressions which describe the behaviour of the components of the simple telephony system. As the examples are developed, visualizations will be used where possible. Each of the slice expressions developed consists of a relatively simple behaviour expression, and a set of process definitions. Note in the examples that all of the process definitions are defined using visualizations; in principle, these process definitions can be generated automatically from the corresponding visual representation.

A simple slice for the telephony example illustrating the basic functionality required for a call connect is shown by the expression `connect` defined in figure 6.2. This slice expression assumes that `user(1)` attempts to connect to `user(2)` and that the connection is successful. Any behavioural specification of the system must include this simple behaviour.

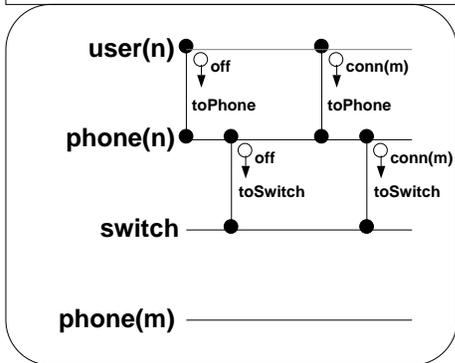
There are however other possible outcomes when `user(1)` attempts to call `user(2)`; for example `user(2)` may be busy, resulting in a busy signal being returned to `user(1)`. This behaviour can be captured by extending the simple slice expression `connect` to represent the possible alternative behaviours. This extension is done in figures 6.3 and 6.4 using the principle of *segmentation* and *construction* described in §3.1.2. Figure 6.3 divides the call connection into the following phases:

- `startConnect(n,m)`. User `n` begins the connection establishment process.
- `establish(n,m)`. The connection is established.
- `busy(n,m)`. The connection is not established because user `m` is busy.

Figure 6.4 constructs these three timeline segments into a single process `extendConnect(n,m)`; after `startConnect` has completed a choice is made and either the connection is established or the user `m` is busy. The slice behaviour specification `extendConnect` can then be defined as the expression `extendConnect(1,2)` which describes the behaviour of `user(1)` connecting to `user(2)` with a possible failure due to `user(2)` being busy. The LOTOS specification is shown in figure 6.5¹. The behaviour expression of the specification is simply an instantiation of the process `extendConnect`. All process definitions were manually generated directly from the corresponding visual representations; the generation of the LOTOS code from the visual notation can be easily automated.

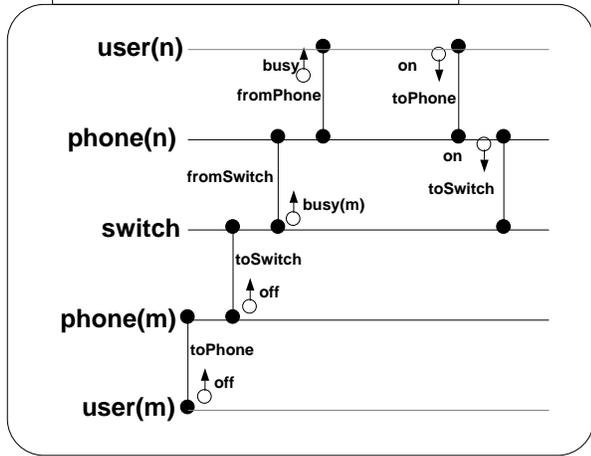
¹ In the LOTOS specifications of this chapter, LOTOS code which can be generated automatically from the visual notations for behaviour are marked with a change bar on the left hand side of the page.

startConnect[...] (n,m: phoneId):exit



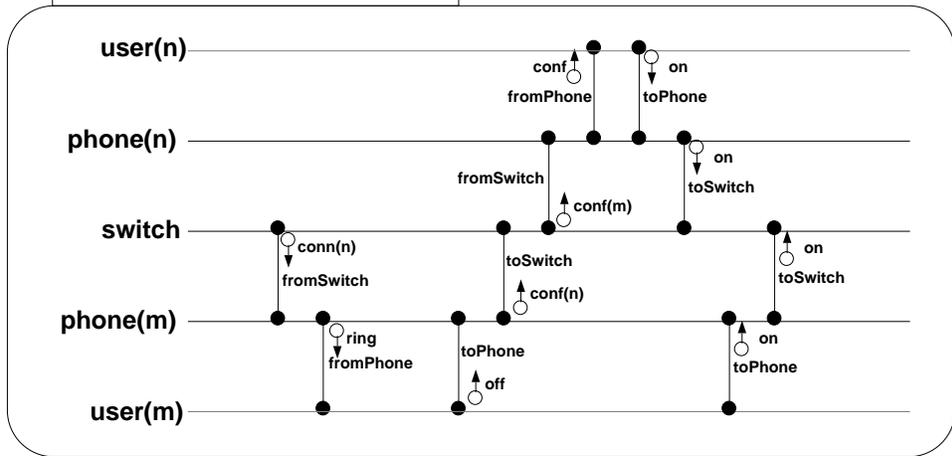
(*User(n) begins the connection establishment process *)

busy[...] (n,m: phonesetId):exit



(* Connection not established - user(m) busy *)

establish[...] (n,m: phoneId):exit



(* connection is established *)

Figure 6.3 Timelines defining different phases of a connection establishment: telephony example.

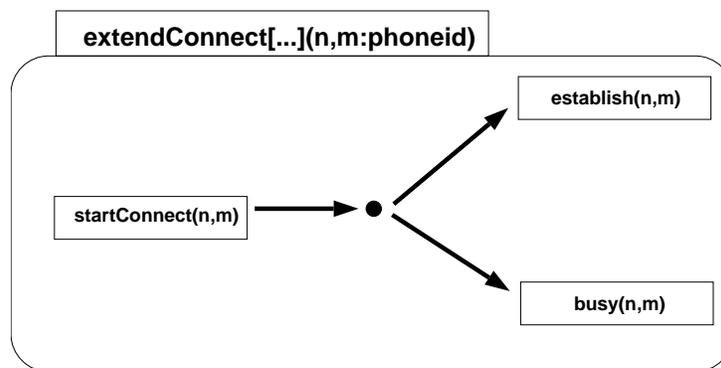


Figure 6.4 Refining the simple call connect by extension.

```

specification extendConnect
  [toPhone,toSwitch,fromPhone,fromSwitch]
  : exit

type signal is phoneid
  sorts signal
  opns
  off,on,conf,ring,busy      :          -> signal
  conf,conn,busy            : phoneid -> signalendtype

type phoneid is
  sorts phoneid
  opns
  1,2 : -> phoneid
endtype

type component is phoneid
  sorts component
  opns
  switch      : -> component
  user        : phoneid -> component
  phone       : phoneid -> component
endtype

(*Behaviour of extendConnect *)

behaviour

extendConnect[toPhone,toSwitch,fromPhone,fromSwitch](1,2)

where

process extendConnect
  [toPhone,toSwitch,fromPhone,fromSwitch]
  (n,m:phoneid):
  exit :=
  startConnect[toPhone,toSwitch](n,m) >>
  (establish[toPhone,toSwitch,fromPhone,fromSwitch](n,m)
  []
  busy[toPhone,toSwitch,fromPhone,fromSwitch](n,m))
endproc

process startConnect
  [toPhone,toSwitch]
  (n,m:phoneid):
  exit :=
  toPhone !user(n) !phone(n) !off;
  toSwitch !phone(n) !switch !off;
  toPhone !user(n) !phone(n) !conn(m);
  toSwitch !phone(n) !switch !conn(m);
  exit
endproc

process establish
  [toPhone,toSwitch,fromPhone,fromSwitch]
  (n,m:phoneid):
  exit :=
  fromSwitch !switch !phone(m) !conn(n);
  fromPhone !phone(m) !user(m) !ring;
  toPhone !user(m) !phone(m) !off;
  toSwitch !phone(m) !switch !conf(n);
  fromSwitch !switch !phone(n) !conf(m);
  fromPhone !phone(n) !user(n) !conf;
  exit
endproc

process busy
  [toPhone,toSwitch,fromPhone,fromSwitch]

```

Figure 6.5 LOTOS specification of **extendConnect**. (Continued ...)

```

        (n,m:phoneid):
        exit :=
toPhone !user(m) !phone(m) !off;
toSwitch !phone(m) !switch !off;
fromSwitch !switch !phone(n) !busy(m);
fromPhone !phone(n) !user(n) !busy;
toPhone !user(n) !phone(n) !on;
toSwitch !phone(n) !switch !on;
        exit
endproc
endspec

```

Figure 6.5 LOTOS specification of `extendConnect`.

The slice expression `connect` of figure 6.2 shows a simple, successful connection. The slice expression `extendConnect` of figure 6.5 shows a connection which may be successful, or which may terminate unsuccessfully. In order to verify that `extendConnect` provides the behaviour of `connect` the *ext* relation of §7.2.1 can be used, and a verification that:

`extendConnect ext connect`

can be performed. Since both behaviour expressions `extendConnect` and `connect` are of finite size, the verification can be done by reachability analysis. (See §7 for a further discussion of the practical issues involved in performing such a verification.)

Concurrent systems have many behaviour patterns executing concurrently. To capture this behaviour requirement it is necessary to be able to specify slice expressions and then to combine these slices expressions in a way which reflects the concurrent nature of the system. For example a slice expression `extendConnect(1,2)` (figure 6.5) defines a connection establishment from `user(1)` to `user(2)` while slice expression `extendConnect(3,4)` defines a concurrent connection establishment from `user(3)` to `user(4)`. To reflect the concurrent nature of systems we would like a mechanism for combining slice expressions to represent this concurrency. As defined in chapter 3 this is refining slice expressions by *slice combination*.

The difficulty in combining slice expressions in parallel is that slice expressions are not necessarily independent of each other, and when combining them the interdependence between slices must be precisely defined. Unfortunately the combination of slice expressions is, in general, a nontrivial process requiring a fair degree of creativity on the part of the designer. If two slices $S1$ and $S2$ are completely independent of each other then we can represent the execution of the two concurrently using the LOTOS interleave operator: $s1 ||| s2$. In the telephony example, the slices `extendConnect(1,2)` and `extendConnect(3,4)` are completely independent of each other and their concurrent behaviour can be expressed as:

```
extendConnect(1,2) ||| extendConnect(3,4)
```

Assume however that we wish to define a slice expression showing `user(1)` calling both `user(2)` and `user(3)`. The naive solution:

```
extendConnect(1,2) ||| extendConnect(1,3)
```

is incorrect since it implies that `user(1)` can be engaged simultaneously in two different calls. In order to combine the two slices correctly, it is necessary to constrain `phone(1)` to a single call at a time.

Constraints can be represented as synchronizing process and defined in LOTOS using a state oriented style. The transition system of figure 6.6 defines a constraint limiting `phone(n)` to one call at a time. In state `idle` the `phone(n)` is not busy. If the handset is lifted, there is a transition to state `busy1` and the `phone(n)` cannot lift the handset a second time until it has been replaced. This constraint can be combined with the two calls to give the following slice expression (the full LOTOS expression is given in figure 6.7):

```
(extendConnect(1,2)      (* user(1) calls user(2) *)
 |||
 extendConnect(1,3))    (* user(1) calls user(3) *)
 | [toPhone] |
 oneCallPerPhone(1)    (* One call at a time *)
```

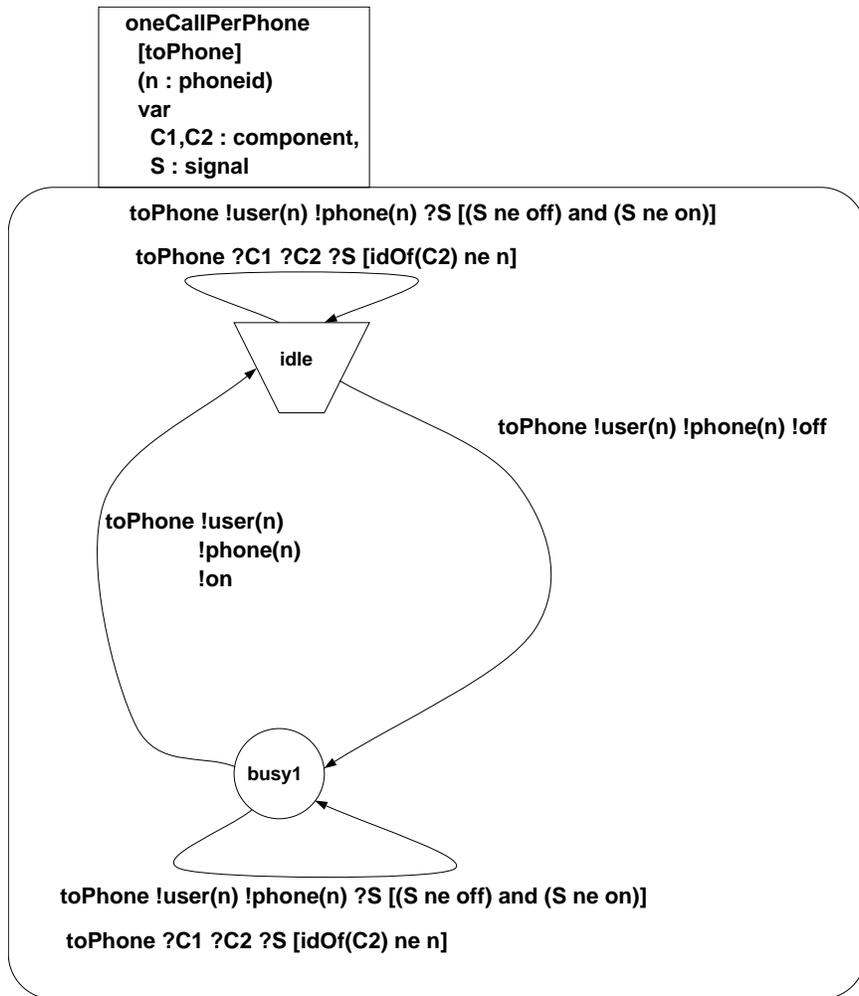


Figure 6.6 State machine defining the constraint that a telephone cannot be engaged in more than one call simultaneously.

```

specification extendConnect
  [toPhone,toSwitch,fromPhone,fromSwitch]
  : exit

library
  Boolean
endlib

type signal is phoneid,Boolean
  (* Define signals between users, phones, and switch *)
  sorts signal
  opns
    off,on,conf,ring,busy      :      -> signal
    conf,conn,busy            : phoneid -> signal
    _eq_,_ne_                 : signal,signal -> Bool
  eqns
    ...
endtype

type phoneid is Boolean
  sorts phoneid
  opns
    1,2,3      : -> phoneid
    _eq_,_ne_ : phoneid,phoneid -> Bool
  eqns
    ...
endtype

type component is phoneid
  sorts component
  opns
    switch      : -> component
    user        : phoneid -> component
    phone       : phoneid -> component
    idOf        : component -> phoneid
  eqns
    forall id:phoneid
      ofsort phoneid
        idOf(phone(id)) = id;
        idOf(user(id)) = id;
endtype

type state is Boolean
  sorts state
  opns
    idle,busy1 : -> state
    _eq_       : state,state -> Bool
  eqns
    forall s:state
      ofsort Bool
        s eq s = true;
endtype

behaviour

```

Figure 6.7 LOTOS slice expression showing **user(1)** calling **user(2)** and **user(3)**. (Continued ...)

```

(* Define two calls occurring concurrently *)
(
  extendConnect[toPhone,toSwitch,fromPhone,fromSwitch] (1,2)
  |||
  extendConnect[toPhone,toSwitch,fromPhone,fromSwitch] (1,3)
)

(* Constrain user(1) to a single concurrent call *)
|[toPhone]|
(oneCallPerPhone[toPhone] (1) [> exit])
where

process extendConnect
  [toPhone,toSwitch,fromPhone,fromSwitch]
  (n,m:phoneid):
  exit :=
  startConnect[toPhone,toSwitch] (n,m) >>
  (establish[toPhone,toSwitch,fromPhone,fromSwitch] (n,m)
  []
  busy[toPhone,toSwitch,fromPhone,fromSwitch] (n,m)
endproc

(* Define process which constrains a phone to a single call*)
process oneCallPerPhone
  [toPhone]
  (n : phoneid):
  noexit :=
  stateMachine[toPhone] (n,idle)

where
process stateMachine
  [toPhone]
  (n : phoneid,
  s : state):
  noexit :=
  [s eq idle] -> ((toPhone !user(n) !phone(n) !off;
  stateMachine[toPhone] (n,busy1))
  []
  (toPhone !user(n) !phone(n) ?S:signal
  [(S ne off) and (S ne on)];
  stateMachine[toPhone] (n,idle))
  []
  (toPhone ?C1:component ?C2:component ?S:signal
  [idOf(C2) ne n];
  stateMachine[toPhone] (n,idle)))
  []
  [s eq busy1] -> ((toPhone !user(n) !phone(n) !on;
  stateMachine[toPhone] (n,idle))
  []
  (toPhone !user(n) !phone(n) ?S:signal
  [(S ne on) and (S ne off)];
  stateMachine[toPhone] (n,busy1))
  []
  (toPhone ?C1:component ?C2:component ?S:signal
  [idOf(C2) ne n];

```

Figure 6.7 LOTOS slice expression showing **user(1)** calling **user(2)** and **user(3)**. (Continued ...)

```

                                stateMachine[toPhone] (n,busy1))
endproc
endproc

process startConnect
  [toPhone,toSwitch]
  (n,m:phoneid):
  exit :=
  toPhone !user(n) !phone(n) !off;
  toSwitch !phone(n) !switch !off;
  toPhone !user(n) !phone(n) !conn(m);
  toSwitch !phone(n) !switch !conn(m);
  exit
endproc

process establish
  [toPhone,toSwitch,fromPhone,fromSwitch]
  (n,m:phoneid):
  exit :=
  fromSwitch !switch !phone(m) !conn(n);
  fromPhone !phone(m) !user(m) !ring;
  toPhone !user(m) !phone(m) !off;
  toSwitch !phone(m) !switch !conf(n);
  fromSwitch !switch !phone(n) !conf(m);
  fromPhone !phone(n) !user(n) !conf;
  toPhone !user(n) !phone(n) !on;
  toSwitch !phone(n) !switch !on;
  toPhone !user(m) !phone(m) !on;
  toSwitch !phone(m) !switch !on;
  exit
endproc

process busy
  [toPhone,toSwitch,fromPhone,fromSwitch]
  (n,m:phoneid):
  exit :=
  toPhone !user(m) !phone(m) !off;
  toSwitch !phone(m) !switch !off;
  fromSwitch !switch !phone(n) !busy(m);
  fromPhone !phone(n) !user(n) !busy;
  toPhone !user(n) !phone(n) !on;
  toSwitch !phone(n) !switch !on;
  toPhone !user(m) !phone(m) !on;
  toSwitch !phone(m) !switch !on;
  exit
endproc
endspec

```

Figure 6.7 LOTOS slice expression showing **user(1)** calling **user(2)** and **user(3)**.

A somewhat more complex problem is how to represent `user(1)` and `user(3)` both placing a call to `user(2)`. The difficulty in combining the two slices `extendConnect(1,2)` and `extendConnect(3,2)` arises because they contain shared instances of interactions. In particular, the single interaction of `user(2)` lifting the handset off the phone can be used by `extendConnect(1,2)` to confirm that the call is established, and by `extendConnect(3,2)` to cause a `busy` signal to be returned to `user(3)`. In order to combine these slice expressions it is necessary to precisely define how these interactions which are shared between different slices are related to each other.

The steps used for combining these slice expressions are described below and illustrated in figure 6.8 through 6.10. The resulting slice expression for the telephony example is shown in figure 6.11.

1. Identify the set I of instances of interactions which are shared between slices $S1$ and $S2$.

In the telephony example the two interactions which are shared between `extendConnect(1,2)` and `extendConnect(3,2)` are:

```
toPhone!user(2)!phone(2)!off    --user(2) picks up phone
toSwitch!phone(2)!switch!off    --phone(2) signals switch
```

2. Construct slice expressions $S1'$, $S2'$. Expressions $S1'$ and $S2'$ are constructed from $S1$ and $S2$ respectively by removing the shared interaction instances I .

In the telephony example the process `extendConnect(n,m)` can be converted to the process `concurrentConnect(n,m)` (figure 6.9) by removing shared interactions:

- `busyPrime(n,m)` is a timeline showing a busy signal being returned to `user(n)`. It is similar to the process `busy(n,m)` of figure 6.3 with the shared interactions identified in step 1 removed.
- `concurrentConnect(n,m)` shows a complete call connection similar to `extendedConnect(n,m)` except the shared interactions identified in step 1 are removed.

3. A synchronizing process *synch* is defined which specifies the relationship between the expressions $S1'$ and $S2'$. The resulting combined slice expression is then given as:

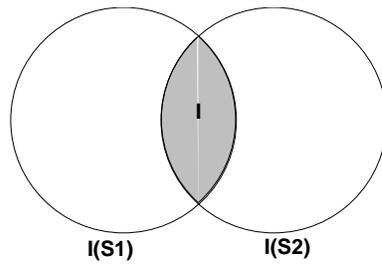
$$(S1' \parallel S2') \llbracket \dots \rrbracket \textit{synch}.$$

The synchronizing process of the example must specify the constraints on `phone(2)` when it is processing two calls. In particular, the constraining process must specify that `phone(2)` will accept a call if it is idle and refuse a call if it is busy. This constraint is specified by the state transition diagram `synchSlices(2)` of figure 6.10. The states of the system are:

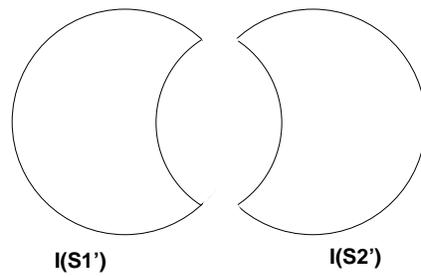
- **idle.** The `phone(m)` is not busy.
- **busy1.** `phone(peer)` has requested a connection to `phone(m)` and is waiting for a response.
- **busy2.** `phone(m)` has confirmed a connection to `phone(peer)`.

The synchronizing process `synchSlices(2)` can be used to combine the expressions `concurrentConnect(1,2)` and `concurrentConnect(3,2)` with the resulting combined slice expression given by:

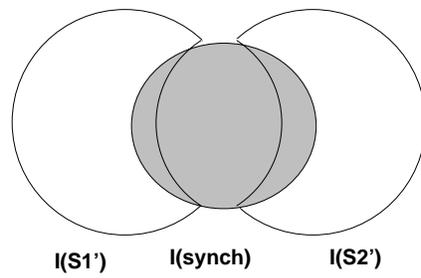
```
twoCalls :=
    (concurrentConnect(1,2)
     |||
     concurrentConnect(3,2)
     |[toSwitch,fromSwitch]|
     synchSlices(2))
```



1. Identify shared interactions I.



2. Construct S1' and S2' with no shared interactions.



3. Construct a synchronizing process such that:
 $(S1' \parallel S2') \text{ synch}$
 represents the composed behaviour.

Figure 6.8 Combining slices.

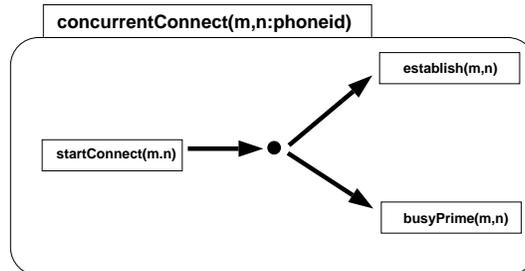
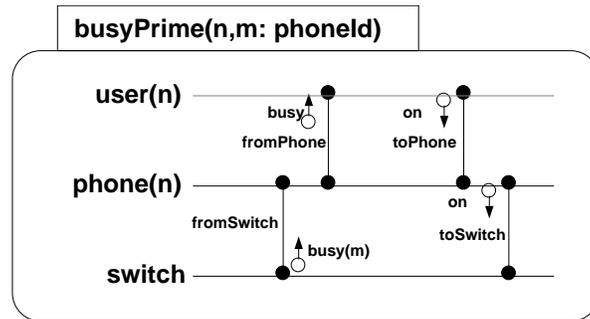


Figure 6.9 Combining slice expressions by removing shared interactions: telephony example.

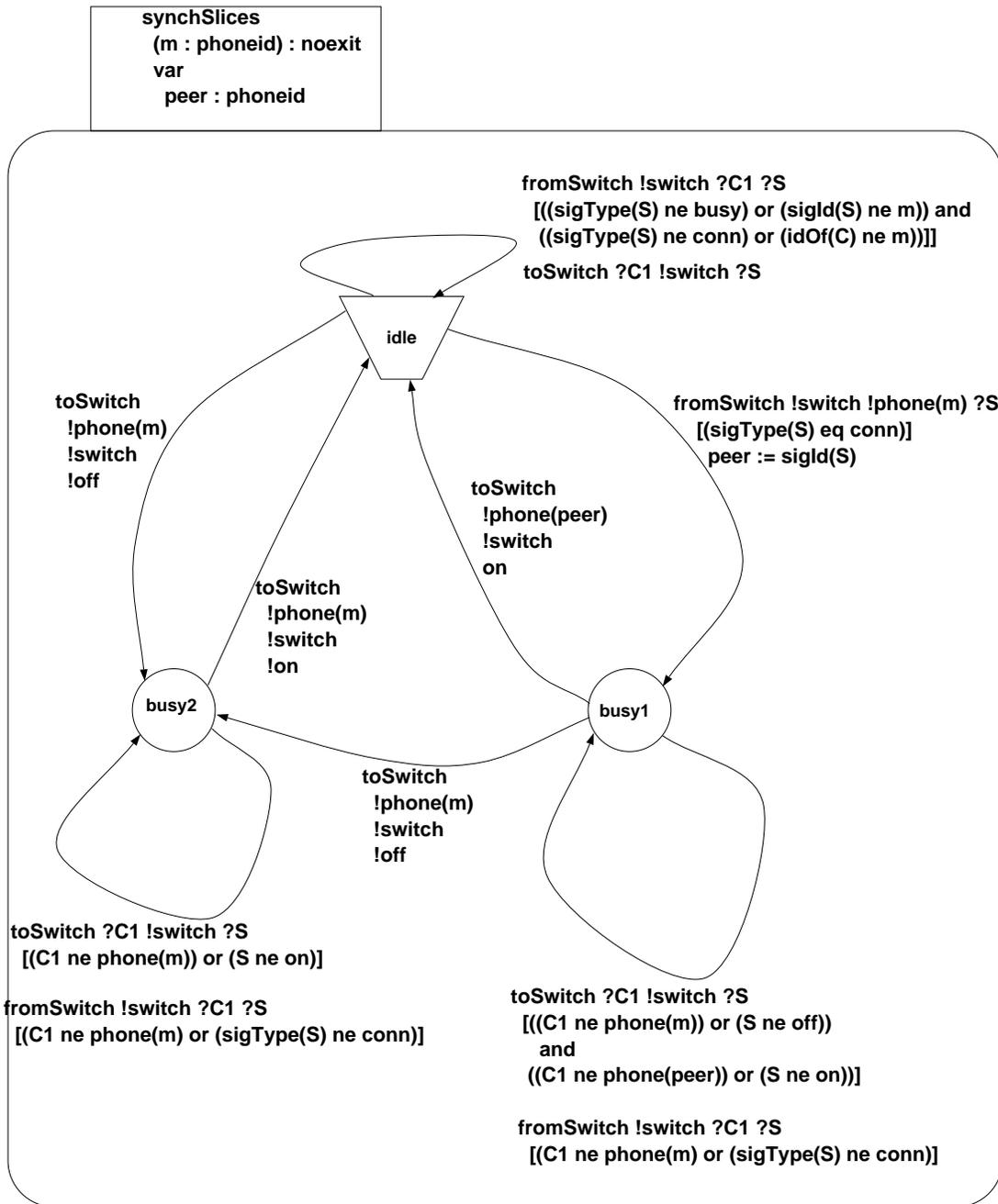


Figure 6.10 State machine used to synchronize slices: telephony example.

```

specification twoCalls
  [toPhone,toSwitch,fromPhone,fromSwitch]
  : exit

  .
  .
  .
behaviour

(
concurrentConnect [toPhone,toSwitch,fromPhone,fromSwitch] (1,2)
|||
concurrentConnect [toPhone,toSwitch,fromPhone,fromSwitch] (3,2)
)
|[toSwitch,fromSwitch] |
(synchSlices[toSwitch,fromSwitch] (2) [> exit)
where

process concurrentConnect
  [toPhone,toSwitch,fromPhone,fromSwitch]
  (n,m:phoneid):
  exit :=
  startConnect[toPhone,toSwitch] (n,m) >>
  (establish[toPhone,toSwitch,fromPhone,fromSwitch] (n,m)
  []
  busyPrime [toPhone,toSwitch,fromPhone,fromSwitch] (n,m))
endproc

process synchSlices
  [toSwitch,fromSwitch]
  (m:phoneid):
  noexit :=
  stateMachine[toSwitch,fromSwitch] (m,idle,m)

where
process stateMachine
  [toSwitch,fromSwitch]
  (m : phoneid,
  s : state,
  peer : phoneid):
  noexit :=
[s eq idle] ->
  ((fromSwitch !switch !phone(m) ?S:signal
  [sigType(S) eq conn];
  stateMachine[toSwitch,fromSwitch] (m,busy1,sigId(S)))
  []
  (toSwitch ?C1:component !switch ?S:signal;
  stateMachine[toSwitch,fromSwitch] (m,idle,peer))
  []
  (fromSwitch !switch ?C1:component ?S:signal
  [((idOf(C1) ne m) or (sigType(S) ne conn)) and
  ((sigId(S) ne m) or (sigType(S) ne busy))];
  stateMachine[toSwitch,fromSwitch] (m,idle,peer)))

[]

```

Figure 6.11 LOTOS expression for **twoCalls**: telephony example. (Continued ...)

```

[s eq busy1] ->
  ((fromSwitch !switch ?C1:component ?S:signal
    [(sigType(S) ne conn) or (idOf(C1) ne m)];
    stateMachine[toSwitch,fromSwitch] (m,busy1,peer))
  []
  (toSwitch !phone(peer) !switch !on;
    stateMachine[toSwitch,fromSwitch] (m,idle,m))
  []
  (toSwitch !phone(m) !switch !off;
    stateMachine[toSwitch,fromSwitch] (m,busy2,peer))
  []
  (toSwitch ?C2:component !switch ?S:signal
    [(idOf(C2) ne peer) or (S ne on)) and
    ((idOf(C2) ne m) or (S ne off))];
    stateMachine[toSwitch,fromSwitch] (m,busy1,peer)))
[]

[s eq busy2] ->
  ((fromSwitch !switch ?C1:component ?S:signal
    [(sigType(S) ne conn) or (idOf(C1) ne m)];
    stateMachine[toSwitch,fromSwitch] (m,busy2,peer))
  []
  (toSwitch !phone(m) !switch !on;
    stateMachine[toSwitch,fromSwitch] (m,idle,m))
  []
  (toSwitch ?C2:component !switch ?S:signal
    [(idOf(C2) ne m) or (S ne on)];
    stateMachine[toSwitch,fromSwitch] (m,busy2,peer)))

endproc
endproc

process startConnect
  [toPhone,toSwitch]
  (n,m:phoneid):
  exit :=
  toPhone !user(n) !phone(n) !off;
  toSwitch !phone(n) !switch !off;
  toPhone !user(n) !phone(n) !conn(m);
  toSwitch !phone(n) !switch !conn(m);
  exit
endproc

process establish
  [toPhone,toSwitch,fromPhone,fromSwitch]
  (n,m:phoneid):
  exit :=
  fromSwitch !switch !phone(m) !conn(n);
  fromPhone !phone(m) !user(m) !ring;
  toPhone !user(m) !phone(m) !off;
  toSwitch !phone(m) !switch !off;
  fromSwitch !switch !phone(n) !conn(m);
  fromPhone !phone(n) !user(n) !conn(m);
  toPhone !user(n) !phone(n) !on;
  toSwitch !phone(n) !switch !on;
  toPhone !user(m) !phone(m) !on;
  toSwitch !phone(m) !switch !on;
  exit

```

Figure 6.11 LOTOS expression for **twoCalls**: telephony example. (Continued ...)

```

endproc
process busyPrime
  [toPhone,toSwitch,fromPhone,fromSwitch]
  (n,m:phoneid):
  exit :=
  fromSwitch !switch !phone(n) !busy(m);
  fromPhone !phone(n) !user(n) !busy;
  toPhone !user(n) !phone(n) !on;
  toSwitch !phone(n) !switch !on;
  exit
endproc
endspec

```

Figure 6.11 LOTOS expression for **twoCalls**: telephony example.

6.3 Generating Component Specifications

Slice expressions are useful as a means of exploring relationships between the behavioural design and the structural design. However in order to allow components to be implemented as separate artifacts it is necessary to have separate behavioural specifications for each component.

Generating a component behavioural specification requires a deep knowledge of the components expected behaviour. A designer deduces the behavioural requirements of a component from a number of sources:

- The requirements of the system in which the component is to be included.
- The set of components from which a system is constructed and their inter-connection topology.
- Descriptions of the functionality required of the component in order to satisfy the system requirements. Among such descriptions are the slice expressions which define the expected behaviour of sets of components interacting to satisfy the system requirements.

Section 5.2 identified two methods for determining behavioural requirements for a component c given a slice expression S :

- *Syntactically.* Using the syntactic transform ϕ_c generate the LOTOS component expression $\phi_c(S)$. Using assumption 3 (page 126), the designer can *extend* $\phi_c(S)$ into a complete behavioural specification for component c .
- *Semantically.* Since LOTOS is an executable specification language, the transition system $\mathfrak{M}(S)$ can be constructed from S and the set $failures(S)$ can be determined. Using theorem 3 (page 140) and assuming that B_c is the behavioural specification of component c , constraints on the set $failures(B_c)$ can be determined by analyzing the set $failures(S)$.

These methods will be illustrated by constructing a component specification for the component `switch` from the slice expression `twoCalls` of figure 6.11.

6.3.1 Generating Component Specifications Syntactically

Expressions for individual components can be syntactically derived from slice expressions. The first step in deriving the component behaviour is to apply the \mathbf{phi}_c transform in order to remove interactions in which component c is not involved. The result of applying the transformation $\mathbf{phi}_{\mathit{switch}}$ to the LOTOS specification $\mathit{twoCalls}$ of figure 6.11 results in the specification switch of figure 6.12. For this example, the transformation was applied manually.

The specification switch can now be executed to determine the behavioural requirements of the switch component. As well, a designer can use this expression as a template for writing a complete behavioural specification for the switch.

```

specification switch
  [toSwitch, fromSwitch]
  : exit

.
.
.
behaviour

(
concurrentConnect [toSwitch, fromSwitch] (1, 2)
|||
concurrentConnect [toSwitch, fromSwitch] (3, 2)
)
|[toSwitch, fromSwitch] |
(synchSlices [toSwitch, fromSwitch] (2) [> exit)
where

process concurrentConnect
  [toSwitch, fromSwitch]
  (n, m: phoneid):
  exit :=
  startConnect [toSwitch] (n, m) >>
  (establish [toSwitch, fromSwitch] (n, m)
  []
  busyPrime [toSwitch, fromSwitch] (n, m))
endproc

process synchSlices
  [toSwitch, fromSwitch]
  (m: phoneid):
  noexit :=
  stateMachine [toSwitch, fromSwitch] (m, idle, m)

where
process stateMachine
  [toSwitch, fromSwitch]
  (m : phoneid,
  s : state,
  peer : phoneid):
  noexit :=
[s eq idle] ->
  ((fromSwitch !switch !phone(m) ?S:signal
  [sigType(S) eq conn];
  stateMachine [toSwitch, fromSwitch] (m, busy1, sigId(S)))

```

Figure 6.12 Component expression for **switch** syntactically derived from **twoCalls**. (Continued ...)

```

[]
(toSwitch ?C1:component !switch ?S:signal;
 stateMachine[toSwitch,fromSwitch] (m,idle,peer))
[]
(fromSwitch !switch ?C1:component ?S:signal
 [((idOf(C1) ne m) or (sigType(S) ne conn)) and
 ((sigId(S) ne m) or (sigType(S) ne busy))];
 stateMachine[toSwitch,fromSwitch] (m,idle,peer)))

[]

[s eq busy1] ->
((fromSwitch !switch ?C1:component ?S:signal
 [(sigType(S) ne conn) or (idOf(C1) ne m)];
 stateMachine[toSwitch,fromSwitch] (m,busy1,peer))
[]
(toSwitch !phone(peer) !switch !on;
 stateMachine[toSwitch,fromSwitch] (m,idle,m))
[]
(toSwitch !phone(m) !switch !off;
 stateMachine[toSwitch,fromSwitch] (m,busy2,peer))
[]
(toSwitch ?C2:component !switch ?S:signal
 [((idOf(C2) ne peer) or (S ne on)) and
 ((idOf(C2) ne m) or (S ne off))];
 stateMachine[toSwitch,fromSwitch] (m,busy1,peer)))

[]

[s eq busy2] ->
((fromSwitch !switch ?C1:component ?S:signal
 [(sigType(S) ne conn) or (idOf(C1) ne m)];
 stateMachine[toSwitch,fromSwitch] (m,busy2,peer))
[]
(toSwitch !phone(m) !switch !on;
 stateMachine[toSwitch,fromSwitch] (m,idle,m))
[]
(toSwitch ?C2:component !switch ?S:signal
 [(idOf(C2) ne m) or (S ne on)];
 stateMachine[toSwitch,fromSwitch] (m,busy2,peer)))

endproc
endproc

process startConnect
[toSwitch]

```

Figure 6.12 Component expression for **switch** syntactically derived from **twoCalls**. (Continued . . .)

```

        (n,m:phoneid):
        exit :=
        toSwitch !phone(n) !switch !off;
        toSwitch !phone(n) !switch !conn(m);
        exit
    endproc

process establish
    [toSwitch,fromSwitch]
    (n,m:phoneid):
    exit :=
    fromSwitch !switch !phone(m) !conn(n);
    toSwitch !phone(m) !switch !off;
    fromSwitch !switch !phone(n) !conf(m);
    toSwitch !phone(n) !switch !on;
    toSwitch !phone(m) !switch !on;
    exit
endproc

process busyPrime
    [toSwitch,fromSwitch]
    (n,m:phoneid):
    exit :=
    fromSwitch !switch !phone(n) !busy(m);
    toSwitch !phone(n) !switch !on;
    exit
endproc

endspec

```

Figure 6.12 Component expression for **switch** syntactically derived from **twoCalls**.

6.3.2 Generating Component Specifications Semantically

The semantic approach to exploring component behaviour through slices can be done by applying theorem 3 (page 140). Application of the theorem allows the designer to ask questions such as:

- Is trace t a behaviour which component c must provide?
- After executing trace t , which interactions can component c offer or refuse?

For example, the behaviour of component `switch` can be derived semantically from the slice expression `twoCalls` of figure 6.11. The designer can ask the question:

- In the initial state (i.e., after the null trace λ) which interactions must the `switch` offer?

To answer this question, all traces t of `twoCalls` such that $t \upharpoonright I(\text{switch}) = \lambda$ must be found. This set consists of the traces:

```
{ $\lambda$ ,  
  toPhone !user(1) phone(1) !off,  
  toPhone !user(2) phone(2) !off}
```

Using theorem 3, the `switch` may refuse an interaction after trace λ only if the slice expression `twoCalls` can refuse the interaction after any of the traces in the above set. Thus in response to the question:

- In the initial state (i.e., after the null trace λ) which interactions must the `switch` offer?

the answer would be

- {toSwitch !phone(1) !switch !off,
 toSwitch !phone(3) !switch !off}

Figure 6.13 is a partial description of the behaviour of `switch` which can be determined from slice expression `twoCalls`. Each entry in the first column is a

Trace of component switch	Interactions which must be offered
λ	toSwitch !phone(1) !switch !off, toSwitch !phone(3) !switch !off
toSwitch !phone(1) !switch !off	toSwitch !phone(3) !switch !off, toSwitch !phone(1) !switch !conn(2)
toSwitch !phone(3) !switch !off;	toSwitch !phone(1) !switch !off, toSwitch !phone(3) !switch !conn(2)
toSwitch !phone(1) !switch !off; toSwitch !phone(3) !switch !off	toSwitch !phone(1) !switch !conn(2), toSwitch !phone(3) !switch !conn(2)
toSwitch !phone(3) !switch !off; toSwitch !phone(1) !switch !off	toSwitch !phone(1) !switch !conn(2), toSwitch !phone(3) !switch !conn(2)
toSwitch !phone(1) !switch !off; toSwitch !phone(1) !switch !conn(2)	fromSwitch !switch !phone(2) !conn(1), toSwitch !phone(3) !switch !conn(2)
toSwitch !phone(1) !switch !off; toSwitch !phone(1) !switch !conn(2); fromSwitch !switch !phone(2) !conn(1)	toSwitch !phone(3) !switch !conn(2), toSwitch !phone(2) !switch !conf(1)
.	.
.	.
.	.

Figure 6.13 List of traces and offers for component **switch** derived from slice expression **twoCalls**.

trace which `switch` must be able to execute. The second column is the set of interactions which `switch` must offer after executing the trace. By examining such behaviour, the designer can determine the behavioural requirements of the components, and then proceed to write specifications (either behavioural or structural) for the components.

6.4 Verifying Component Specifications

As the component specifications are developed it is necessary to verify them for correctness. Given a slice expression S , this can be done by verifying that the behavioural specification B_c of component c is capable of providing all the behaviour implied by the slice expression S (see §5.4). Thus each slice expression becomes a set of test cases for the component specifications.

If B_{switch} is a behavioural specification for the `switch`, there are three ways that the slice `twoCalls` (figure 6.11) can be used as a basis to test B_{switch} :

- Using assumption 3 (page 126), verify the relation:

$$B_{switch} \text{ ext } \phi_{switch}(\text{twoCalls})$$

- Use $traces(\text{twoCalls}) \upharpoonright I(\text{switch})$ as a set of test cases for B_{switch} (theorem 2, page 139).
- Use $failures(\text{twoCalls})$ as a basis for producing a set of test cases for B_{switch} (§5.4).

Verifying the relation $B_{switch} \text{ ext } \phi_{switch}(\text{twoCalls})$ requires first generating the LOTOS expression $\phi_{switch}(\text{twoCalls})$; the resulting expression was shown in figure 6.12. The complexity in verifying the *ext* relation is dependent on the number of states in the transition system model of $\phi_{switch}(\text{twoCalls})$. Issues involved in proving this relation are discussed further in §7.2.1.

When using the set $traces(\text{twoCalls}) \upharpoonright I(\text{switch})$ as a basis for verifying B_{switch} it is first necessary to generate the set $traces(\text{twoCalls})$ and then project these traces through the events of `switch`. The first column of the table of figure

6.13 is a partial listing of traces of `switch` generated in this way. Each of these traces can be considered as a test case for B_{switch} . The number of test cases generated depends on the number of maximal traces for `switch` which can be generated from `twoCalls`. For the simpler expression `extendConnect` (figure 6.5), it can be verified by looking at figures 6.3 and 6.4 that the `switch` will engage in a trace of length 7 when a connection is correctly established, and a trace of length 5 when a busy signal is returned. The expression `twoCalls` is an interleaving of two instances of `extendConnect`, restricted by the process `synchSlices`. Since the number of ways two sequences can be interleaved grows exponentially with the length of the sequences, generating all traces from a slice expression which is the concurrent combination of a number of slice expressions could lead to an intractable number of traces being generated. This problem is discussed further in Chapter 7.

For deterministic slice expressions (such as all the examples of this chapter and Appendix D), testing based on *failures* and testing based on *traces* are equivalent.

As shown in §5.4, if the original slice expression contained nondeterminism then the composition of the component behaviours may not be an *extension* of the slice expression, even if all the component behaviour specification pass the above tests. The problem is that the intention of the designer in constructing the slice expression is that the nondeterminism of the slice will result in one of the components being nondeterministic; if the nondeterminism appears in more than one component then interactions can be *refused* by the composition of the component behaviour specifications which could not be refused by the slice expression S . The method for detecting such problems is to compose a set of component behaviour specifications into a behaviour expression B_D according to the system structure (this set could be a subset of the total components of the system). Verification then proceeds by comparing the set $failures(B_D)$ with the set $failures(S)$ as described in §5.4; if there is a *failure* of B_D which violates the conditions imposed by $failures(S)$ then one of the component behaviour

specifications of B_D is incorrect. For example, in the telephony system, assume the designer has constructed behaviour specifications B_{switch} and $B_{phone(1)}$ for components `switch` and `phone(1)`. The system can now construct expression $B_{\{switch,phone(1)\}}$ as:

$$\begin{aligned}
 B_{\{switch,phone(1)\}} &:= B_{switch} \parallel \text{default}(switch) \\
 &\parallel \\
 &B_{phone(1)} \parallel \text{default}(phone(1))
 \end{aligned}$$

The *failures* of this expression can then be compared to *failures*(`twoCalls`) in order to determine whether B_{switch} and $B_{phone(1)}$ are incorrect. (Note that since slice expression `twoCalls` is deterministic, it is not necessary to perform such a test; using only traces is sufficient.)

6.5 Component Decomposition

Components can be viewed either as monolithic units which are implemented as a single entity, or as having an internal structure composed of subcomponents. In the former case, the development path after generating the component specification B_c is to implement the component based on this behavioural specification. However if a component is complex and constructed from subcomponents, a designer can repeat the development process for component c using the specification B_c as the requirements specification. The development process then includes the creation of a structural design, slice expressions and component expressions for the internal components, and analysis of the design for verification purposes.

In the simple telephone system example, the structure was specified in figure 6.1. Assume that the switch is a distributed component constructed from two subcomponents (figure 6.14).

One issue that arises is how the behavioural expressions developed at one level of the structural decomposition (e.g., at the level of figure 6.1) can be applied to

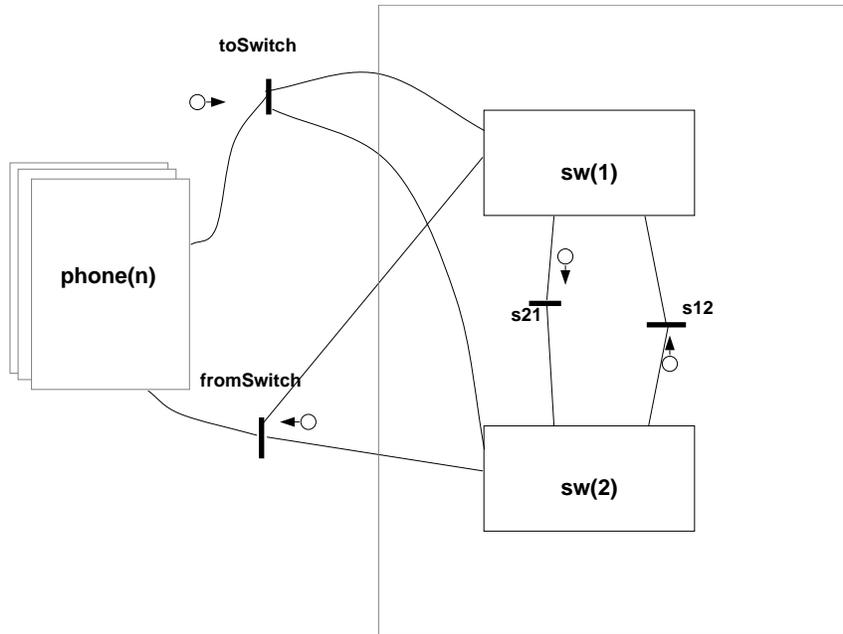


Figure 6.14 Decomposed switch component.

the analysis of design at another level of structural decomposition (e.g., at the level of figure 6.14). The two problems identified in §4.2.3 were:

- *Interactions internal to a component.* At lower levels of the structural hierarchy, there may be interactions internal to a component which were not visible at higher levels of the design hierarchy.
- *New component names introduced by the decomposition.* An interaction involving component c specified at one level of the design hierarchy may involve a subcomponent of c at a lower level of the design hierarchy.

LOTOS includes a powerful encapsulation technique for *hiding* events. By this method the behaviour of the `switch` component can be represented as the composition of the behaviours of `sw(1)` and `sw(2)` with the IPs `s21` and `s12` hidden. For example, if the behaviours of `sw(1)` and `sw(2)` are given by $B_{sw(1)}$ and $B_{sw(2)}$, then their composed behaviour is given by:

$$\begin{aligned}
 B_{switch} &:= \text{hide } s21, s12 \text{ in} \\
 &\quad B_{sw(1)} \parallel \text{default}(sw(1)) \\
 &\quad \parallel \\
 &\quad B_{sw(2)} \parallel \text{default}(sw(2))
 \end{aligned}$$

When using the slice style as presented in this research, a designer specifies behaviour in terms of interactions, where an interaction includes both the IP and the component identifiers synchronizing at the IP. During decomposition, what appears as a single component from one structural viewpoint is actually a collection of components from a finer grained structural viewpoint. As described in §4.2.3 this may require that the interactions at the finer grained structural viewpoint are renamed in order that they can be verified against slice expressions created with the coarser grained structural viewpoint. For example at the structural

level of figure 6.14 the interaction:

`toSwitch!phoneset(1)!sw(1)!off`

can be transformed to:

`toSwitch!phoneset(1)!switch!off`

in order to verify it against slice expressions created at the structural level of figure 6.1. Such a renaming is defined by the structure renaming function Δ_{switch} :

$$\Delta_{switch}(c) = \begin{cases} switch & \text{if } c = sw(1) \text{ or } c = sw(2) \\ c & \text{otherwise} \end{cases}$$

By applying the structure renaming function Δ_{switch} to the interactions of B_{switch} , it is possible to test B_{switch} using the slice expression `twoCalls`.

Chapter 7: Practicality of Slice Techniques

This thesis has developed techniques for the design of concurrent systems and applied these techniques in a number of examples and case studies. This chapter looks at the practicality of the techniques, discusses how well the techniques can be applied manually, where tool support could assist a designer, and the possible limitations of analysis and verification.

7.1 Design Capture

Design capture is the process by which a designer's decisions are represented and recorded. This research has developed formal representations for structural and behavioural design, and visual notations which assist the designer in making and recording design decisions during the design process.

For all the designs developed as part of this research, the structural design was developed using the visual notation of §4.1.1.1. It was not found to be necessary to generate the explicit formal structural representation; in general all the behavioural specifications could be developed by referring only to the visual structural notation. For the examples of this thesis, generating the formal representation from the visual structural notation was a straightforward (but tedious) task that could easily be automated.

Behavioural design was developed using a combination of visual notations as well as directly in LOTOS. Within the telephony example of Chapter 6 and the case study of Appendix D all process definitions can be (and were) represented visually. Timelines were used to represent sequential behaviour. These simple sequential slices were extended by means of segmentation and construction, operations which could also be performed visually using the notation of §4.2.2. Combining slices to represent concurrent behaviour required the definition of synchronizing processes; all such processes were defined visually

as state transition systems using the notation of §4.2.2. Thus very little explicit programming of LOTOS behaviour expressions was required; it is feasible to construct tools which allow most of the LOTOS to be generated automatically from the visual representation. In the examples of Chapter 6, the parts of the LOTOS specifications which could have been generated automatically from the visual notation are marked with a side bar on the left hand side of the page. It can be seen that all process definitions in the specifications can be generated from the visual notation.

The following are some qualitative observations about the applicability of the visual notations, and how they were used during this research:

- Timelines were most useful for representing simple behaviour sequences through sets of components. As more complex slices were created through extension, manually segmenting and constructing timelines was difficult to perform without tool support. However, the visual representation was found to be very helpful and useful, and a tool to assist a designer in slice extension is feasible and would be highly desirable.
- State transition diagrams were used to represent synchronizing processes when slices were refined by combining slices concurrently. For this purpose, the author found the visual notation extremely useful. Even without a tool available, the generation of these processes was facilitated by visually representing them as state transition diagrams and then manually transforming the visual representation into a LOTOS process. Once a tool had been developed [57] the synchronizing processes were automatically generated from the visual notation.

Although the examples of this thesis were performed with minimal tool support, there clearly are areas where tools could greatly enhance the ability of the designer. Specific tools which could be used during design capture include the following:

- *Structural design capture.* A graphical editor could be used to facilitate the entry of structural designs. A number of consistency checks could be performed on the designs, such as verifying that all IP sides are connected to components, IP characteristics are consistent across structural levels, etc.
- *Behavioural design capture.* Simple slice expressions can be visually represented either as timelines or by tracing a slice directly on the structural design diagram (as is done in many examples of this thesis). Graphical editors would permit a designer to construct these slice expressions visually.
- *Refining slices.* Slices are refined by extension or by combination. Slice extension can be supported by a tool which allows a designer to cut a timeline into segments, and to construct a slice with timeline segments using the notation of §4.2.2. Slice combination can be supported by a tool which assists a designer in identifying locations where slices interact with each other (§6.2), and by a graphical editor for defining synchronizing processes as state transition diagrams with automatic generation of the LOTOS code (§4.2.2).
- *Executing slice expressions.* Since LOTOS is an executable specification language, a LOTOS interpreter can be used to execute the slice specifications in order to verify their correct behaviour. The ISLA interpreter [30,62] was used for this purpose and found to be invaluable for generating and testing slice expressions.

7.2 Design Analysis and Verification

This research has identified a number of areas where analysis and verification techniques can be applied to assist a designer during the design process. This section discusses the practical application and limitation of techniques in the following areas:

- *Proving the ext relation* (§7.2.1). The design process assumed that the *ext* relation existed between different behaviour expressions. One verification

which can be used during the design process is to prove that the *ext* holds in these cases. Specifically:

- If slice S_2 is a refinement of slice S_1 then $S_2 \text{ ext } S_1$;
 - If B_c is a designer generated specification for component c and S is a slice expression then $B_c \text{ ext } \phi_c(S)$.
 - If $B_{\langle IP, C \rangle}$ is a resource oriented specifications and S is a slice expression, then $B_{\langle IP, C \rangle} \text{ ext } S$.
- *Extracting component behaviour from slices* (§7.2.2). Given that a designer has created a structural design and a set of slice expressions, it is necessary to construct an individual component specification B_c for each component c . This can be done by extracting the behaviour requirements of component c from slice S and using these requirements to assist in the development of B_c , or to verify that B_c satisfies the requirements imposed by S . Analysis tools can extract the behaviour of c implied by S and verify that B_c satisfies these behavioural requirements. Also, by comparing the component closure $\phi(S)$ to S an analysis of a system for critical races can be performed.

7.2.1 Verifying the *ext* Relation

Due to the complexity and size of most behavioural specifications, proving the *ext* relation between two behaviour expressions is practical only if the proof can be automated.

If two transition systems B_1 and B_2 are both finite (i.e., contain a finite number of states and a finite number of transitions) then a decision procedure for determining whether $B_1 \text{ ext } B_2$ exists. Verifying $B_1 \text{ ext } B_2$ is related to proving testing equivalence [3,50]. Proving testing equivalence between finite transition systems is known to be a PSPACE-complete problem [50] and it is therefore unlikely that a polynomial algorithm for verifying testing equivalence exists. Since B_1 is testing equivalent B_2 if and only if $B_1 \text{ ext } B_2$ and $B_2 \text{ ext } B_1$ [7]

a polynomial algorithm for proving *ext* would result in a polynomial algorithm for proving testing equivalence, an unlikely prospect.

An algorithm for verifying testing equivalence between finite transition systems is developed by Bolognesi and Caneve in [3]. It seems probable that such an algorithm can be adapted to verify the *ext* relation. Although the algorithm of [3] is exponential in time complexity, the step of the algorithm which causes the exponential complexity involves transforming a nondeterministic transition system to a deterministic transition system. If the behaviour expressions are deterministic (as is the case with all the slice expressions generated in the telephony example of Chapter 6 and in the case study of Appendix D), then a polynomial algorithm is likely attainable. The actual details of the algorithm are left as an area for further research.

For deterministic, finite slice expressions, the growth in complexity of proving the *ext* relation is polynomial in the number of states. Given two slice expressions with n and m states, combining these expressions in parallel will result in a slice expression with at most nm states. In the example of Chapter 6, the slice expression `extendConnect` (figure 6.5) illustrating a call with a possible result of connection establishment or busy signal returned contained approximately 20 different states. The projection of this expression onto the component `switch` by means of the transform ϕ_{switch} results in an expression containing approximately 11 states. Thus representing n concurrent calls by combining these expressions will result in a slice expression which contains at most 20^n states or 11^n states for the system or `switch` respectively.

In the case study of Appendix D, a `switch` component was constructed from seven components. The most complex slice expression representing a call connection between two phone sets involved approximately 60 states. Thus in the worst case, n concurrent calls are represented by a slice expression with 60^n states. Clearly in this case, the value of n must be rather small for a complete verification of the *ext* relation to occur by reachability analysis.

7.2.2 Extracting Component Behaviour from Slices

Once a designer has specified a structural design and used slices to specify the behaviour patterns involving many components, it is necessary to generate behavioural specifications for the individual components. Although the process of writing the specification remains primarily a manual operation, slices can assist the designer in the following ways:

- From the slice expression, the behaviour of an individual component can be extracted (§5.2).
- It can be verified that the expression representing a component's behaviour satisfies all the constraints imposed by a slice expression (§5.4).

Two approaches were defined for extracting component behaviour from slice expressions:

- The syntactic approach where each LOTOS slice expression S is transformed into LOTOS expression $\phi_c(S)$ representing the behaviour of component c (§5.2.1).
- The semantic approach where given the slice expression S the transition system model $\mathfrak{M}(S)$ is constructed, and from this a model of behaviour for component c is determined (§5.2.2).

The transform ϕ_c as described in §5.2 and Appendix E provides a means for syntactically deriving component behaviour from a slice expression. Application of the transform is straightforward and can be done manually, as was done in applying the transform to the slice expressions of Chapter 6 and Appendix D. Results obtained indicated that the transform is relatively easy to apply manually and that there are no practical problems in automating most of the transformation. The transform cannot be fully automated as some input may be required by the designer to determine possible values of data expressions which represent

component names. A LOTOS interpreter can be used to execute the expression $\phi_c(S)$ and explore the behaviour of component c .

Once the designer understands the behaviour of the component, the complete component specification B_c can be generated. The expression $\phi_c(S)$ can be used as a template which the designer can extend into the complete component specification B_c . The experience gained during this research indicates that for most components the changes required to $\phi_c(S)$ are complex and the extensions required are not always obvious; as a result viewing $\phi_c(S)$ as a template for constructing B_c is not always feasible. A more reasonable approach is to use $\phi_c(S)$ to explore the behaviour of c , and then manually generate the specification B_c . It can then be verified that $B_c \text{ ext } \phi_c(S)$.

The component specifications B_c can be composed into a resource oriented expression $B_{\langle IP, C \rangle}$ representing the overall system behaviour. This expression can be executed using an interpreter in order to explore the system behaviour. Using existing tools, there were practical limitations on the size of a resource oriented specification which could be executed due to efficiency problems; the problems encountered are discussed further in §7.3.

One analysis described in this thesis is a means of detecting potential critical races by comparing the component closure $\phi(S)$ with the slice expression S (§5.3). Due to the complexity of such an analysis, a manual approach is not feasible and an automated tool is required in order to make the analysis practical. The analysis requires a comparison of the set $traces(\phi(S))$ with the set $traces(S)$. The computational complexity depends on the size of the set $traces(\phi(S))$. Unfortunately, if slices S_1 and S_2 are combined in parallel into the expression $S_1 \# S_2$, the size of the set $traces(S_1 \# S_2)$ grows exponentially with respect to the size of the sets $traces(S_1)$ and $traces(S_2)$, and with the length of the traces. This results in the size of the set $traces(\phi(S))$ growing very quickly beyond the limits of computational feasibility. Although a complete analysis of the set $traces(\phi(S))$

is not feasible, there is reason to be optimistic that a useful analysis of the set to detect serious races is feasible:

- To determine if $\phi(S)$ is serious race free it is necessary to examine each element of the set $traces(\phi(S))$. However, to determine if $\phi(S)$ contains a serious race it is only necessary to examine the elements of $traces(\phi(S))$ until a serious race is found. Assuming a generate-and-test strategy is used, this involves generating traces of $\phi(S)$ and testing for a serious race. The feasibility of the approach then depends not on the total number of elements in the set $traces(\phi(S))$, but rather on how many traces must be generated before one with a serious race is encountered. Looking at the gas station example (figure 5.19, page 153), approximately one half of the traces of the component closure contain the serious race. Therefore, assuming that traces are generated in a random fashion, the average number of traces which must be generated before the serious race is found is actually quite small.
- Since LOTOS models concurrency by means of interleaving rather than a partial ordering, interactions which have no temporal relationship are interleaved in some arbitrary order. Thus the set $traces(\phi(S_1\#S_2))$ will contain a large number of traces which are differentiated only by the various interleavings of unrelated events. If S_1 and S_2 do not pass through any of the same components then they can have no effect on each other and it is not necessary to look at all possible interleavings. For example, the slice expression `twoCalls` (figure 6.11, page 183) represented a call from `user(1)` to `user(2)` and from `user(3)` to `user(2)`. These calls cross over only within the components `switch` and `phone(2)`. Interactions not involving these components can be interleaved in any manner without any effect on whether a critical race will or will not be found. Thus the actual number of traces of $\phi(S_1\#S_2)$ which must be generated to verify that it is serious race free may be much less than the actual number of elements in $traces(\phi(S_1\#S_2))$. Techniques to determine which traces must

be generated to verify that the system is serious race free are left as areas of future research.

The semantic approach to extracting the component behaviour of c from slice S is to generate the model $\mathfrak{M}(S)$ and from this model determine the requirements on component c . This can be done either by looking at the set $traces(S)$ or the set $failures(S)$ and from this determining requirements on the set $traces(B_c)$ or the set $failures(B_c)$. The difficulty in generating and manipulating these sets indicates that a manual analysis is not feasible for larger systems and a tool to assist the designer is required. The computational feasibility of such an approach then depends on the following:

- The number of *traces* or *failures* of expression S .
- The complexity in computing the *traces* or *failures*.

Generating traces of a component from the traces of a slice expression is relatively easy to do and requires little more than an efficient LOTOS interpreter. In the case study of Appendix D the most complex slice expression representing a call connection is the slice `branchSwitch` (figure D.10, page 257). The traces of this expression contained a total of 9 maximal length traces (i.e., traces which are not proper prefixes of any other trace); these traces range in size from 2 interactions to 36 interactions. Clearly, very little computing power is required to generate these traces and to use them to determine the behaviour of individual components.

However, since parallelism is represented by interleaving, once slice expressions are combined in parallel the number of maximal length traces can grow exponentially. Although in worst case the number of traces can grow exponentially as slices are combined concurrently, in practice the growth in the number of traces of interest should be significantly less. In the telephony example of this chapter, the slice expression `twoCalls` (figure 6.11, page 183) represents two calls combined in parallel. Where these calls do not interfere with each other,

the interleavings of the interactions represent completely independent sequences of events; thus it is not necessary to look at all these interleavings. Where more thorough analysis is required, is at the points where the calls interfere with each other; but this is exactly the behaviour captured by the synchronizing process `synchSlices`. Therefore it is only the behaviour represented within the process `synchSlices` which must be analyzed extensively when using `twoCalls` as a basis for extracting component behaviour. The state transition diagram for `synchSlices` contains only three states and eight transitions, so the problem appears to be much more feasible.

For deterministic slice expressions it is not necessary to investigate the *failures* as well as the *traces*; the behaviour requirements can be completely captured by the *traces*. All the slice expression of Chapter 6 and Appendix D are deterministic and therefore it is not necessary to look at failures. If failures are used as a basis for determining component behaviour, generating the set $failures(S)$ may result in feasibility problems as discussed in §7.2.1.

7.3 Design Execution

The semantics of LOTOS is defined operationally: given a LOTOS behaviour expression B the operational semantics define the transition system $\mathfrak{M}(B)$ which is the model of B . A LOTOS interpreter is a tool which given the LOTOS specification B can construct the model (or part of the model) $\mathfrak{M}(B)$. This permits a designer to execute the specification and to explore the behavioural model.

During the course of this research, the interpreter ISLA [30,62], developed at the University of Ottawa, was used to explore LOTOS behavioural specifications. The interpreter was used extensively and was invaluable for validating behavioural specifications.

The primary problem encountered in the use of the ISLA interpreter was the performance of the interpreter with resource oriented specifications involving a large number of components. This performance problem became evident during

the case study described in Appendix D where the time required to interpret one interaction in a design involving approximately twenty components became prohibitively long (i.e., executing a trace involving 10 or 15 interactions required hours to complete).

Since the performance problem was observed only in large resource oriented specifications, it is likely that the problem results from the way in which resource oriented specifications are structured and interpreted. In order to explore the problem of resource oriented specifications in more detail, a LOTOS resource oriented specification of the design structure of figure 7.1 was written. The design structure consists of a pool of n components $1_{\text{oop}}(1)$ to $1_{\text{oop}}(n)$ which synchronize with each other at the interaction point `event`. Each component $1_{\text{oop}}(k)$ loops forever trying to synchronize with component $1_{\text{oop}}(k-1)$ and $1_{\text{oop}}(k+1)$. The complete LOTOS specification is given in figure 7.3.

Figure 7.2 shows the performance of the ISLA interpreter for different values of n . The left column of the table shows the number of components $1_{\text{oop}}(k)$ which were created. The corresponding value in the right column is the time it took for the interpreter to execute a single interaction and then determine the list of enabled events. The tests were run with a maximum inference width of twenty events and an inference width of sixty events. The platform used was a Sun SparcStation 2, with 16MB of memory and 52MB of swap space. As can be seen, the time required for the ISLA interpreter to execute a single step becomes prohibitively large very quickly. In performing the case study of Appendix D the times for interpreting an event were large enough that there were practical limitations in the length of event sequences which could be executed during a verification.

During the case study, the performance of the interpreter was improved significantly by tuning the interpreter and the specification in different ways, for example:

- removing the *default* processes where possible.
- limiting the inference width of the ISLA interpreter.

- reordering processes in the LOTOS specification.

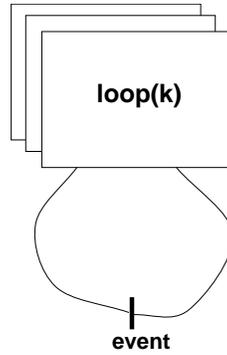


Figure 7.1 Design structure used to determine performance of ISLA interpreting a resource oriented specification.

Number of Components	Time required to interpret a trace of length 1 (min:sec)	
	inference width=20	inference width=60
5	0:05	0:07
10	0:25	0:44
11	0:50	1:41
12	1:03	2:17
13	2:40	8:54
14	9:12	>15:00
15	>15:00*	—

* Warning messages indicated machine was out of memory.

Figure 7.2 Performance of ISLA interpreting a resource oriented specification.

```

specification seed[event](n:Nat) : noexit

library
  NaturalNumber
endlib
type    Component is NaturalNumber

  sorts    Component

  opns     component    : Nat -> Component
           idOf        : Component -> Nat

  eqns     forall id:Nat
           ofSort Nat
           idOf(component(id)) = id;
endtype

behaviour

grow[event](n,0,Succ(0),n)

where
  (*process grow creates the n+1 component processes. *)
  process grow
    [event]
    (lower,n,upper,limit : Nat):
    noexit :=
    ([n eq limit] -> (loop[event](lower,n,0) ||| default[event](n)))
    []
    ([n eq 0] and (n ne limit)) ->
      ((loop[event](lower,n,upper) ||| default[event](n) )
      ||
      grow[event](0,Succ(0),Succ(upper),limit))
    []
    ([n ne 0] and (n ne limit)) ->
      ((loop[event](lower,n,upper) ||| default[event](n) )
      ||
      grow[event](Succ(lower),Succ(n),Succ(upper),limit))
  endproc

  (*process loop is the behaviour of a single component. *)
  process loop
    [event]
    (lower,n,upper : Nat):
    noexit :=
    (event !component(n) ?C:Component [idOf(C) eq upper];
     loop[event](lower,n,upper))
    []
    (event ?C:Component !component(n) [idOf(C) eq lower];
     loop[event](lower,n,upper))
  endproc

  process default
    [event]
    (n : Nat):
    noexit :=
    event ?C1:Component ?C2:Component
      [(idOf(C1) ne n) and (idOf(C2) ne n)];
     default[event](n)
  endproc
endspec

```

Figure 7.3 LOTOS specification used to determine performance of ISLA interpreting a resource oriented specification.

7.4 Implementation

This research has developed design techniques which are applicable to developing component behavioural specifications as LOTOS behaviour expressions. How do these specifications correspond to actual implementations, particularly software implementations?

Given that a LOTOS specification of a component exists, there are a number of possible techniques which can be used to develop the actual component implementation:

- *LOTOS compilers.* Given that many LOTOS specifications are executable, it is conceivable that the LOTOS component specifications are in fact the implementation. With an efficient LOTOS compiler, an executable object can be generated directly and automatically from the LOTOS specification.
- *LOTOS translators.* A translator can be used to transform the LOTOS specification into an implementation language such as Ada or C.

A complete discussion of the issues involved in developing LOTOS compilers and translators is beyond the scope of this thesis.

7.5 Chapter Summary

During the course of this research, a number of techniques were developed which can be applied to the design of concurrent systems. From the experience gained, the following conclusions are drawn:

- Many of the techniques for visual design capture can be effectively applied without tool support.
- Tools for design capture would facilitate a designers task and enhance the designer's capabilities. These tools include graphical editors for visually representing structural and behavioural design, as well as tools to assist a designer in refining slices.

- As a minimum tool support, the use of a LOTOS interpreter is essential. Further research is required to permit the efficient interpretation of large resource oriented specifications.
- Most of the analysis and verification techniques require automation in order to be effectively applied to large specifications. More empirical case studies, and research into verification techniques are required. As well, further research is required into problems of intractability, particularly when slices are combined in parallel.

Chapter 8: Conclusions

The objective of this research is to formalize specific techniques which can be applied to concurrent system design. This chapter summarizes the specific contributions which have been made as a result of this research, draws some conclusions about the applicability of the techniques, and identifies areas which require further research.

8.1 Contributions

While investigating and formalizing techniques which can be used in concurrent system design, a number of contributions were made by this research. This section identifies the specific contributions.

8.1.1 Formalization of a Slice Technique

Designing concurrent systems by thinking in terms of ‘slices’ is a method that is often used informally. The major thrust of this research involved formalizing the ‘slice style’ and developing techniques which can be applied to the design process. The contributions towards this goal can be summarized as follows.

Representing slice behaviour as LOTOS expressions. In order to express behaviour patterns involving sets of components, the formal language LOTOS was integrated with a structural design representation and used to express slice behaviour expressions (§4.2.1.1 and §4.2.1.2). Since slice expressions are not complete representations of behaviour, but rather partial descriptions, it is necessary to define a relation between the slice expression and the complete system behaviour so that it can be verified that the system actually does exhibit the behaviour specified by the slice. The relation used is *ext* (§5.1) whereby a system behaviour B is correct relative to a slice S only if $B \text{ ext } S$.

Developing slice expressions. Slices are developed in an incremental manner. Two types of slice refinement were identified in chapter 3:

- *Slice extension.* Beginning with a simple slice S , the designer creates slice expression S' by adding new behaviour to S . Extending slices is accomplished by segmenting and constructing new slices as described in §3.1.2 and illustrated in Chapter 6 and Appendix D.
- *Slice combination.* Beginning with two slices S_1 and S_2 the designer combines these into expression $S_1\#S_2$. Techniques for synchronizing combined slice expressions are described in §3.1.2 and illustrated in Chapter 6 and Appendix D.

Developing component behaviour specifications. In order to develop a component as a separate implementation artifact it is necessary to have a behaviour specification for each individual component. Two methods for extracting component behaviour from slice expressions are identified (§5.2):

- *Syntactically* (§5.2.1). Given a LOTOS slice expression S and a component c , a transformation ϕ_c is defined such that $\phi_c(S)$ represents the behavioural requirements of c , expressed in LOTOS, implied by slice S . Once the component specifications have been syntactically extracted from the slice expression S , they can be composed according to the method of §4.2.1.2 into the component closure $\phi(S)$ to assist a designer in detecting critical races in the design (§5.3).
- *Semantically* (§5.2.2). Given a slice expression S one method for determining the behaviour of component c implied by S is to execute S thereby constructing its transition system model, and then extracting from the transition system model the behaviour of component c . Theorems 2 and 3 (pages 139 and 140) provide a basis for such a semantic approach to determining the behaviour of component c given slice S .

Once the behaviour of component c has been extracted from slice S , this can be used as a means of assisting the designer in generating the component behaviour specification B_c , or verifying that B_c satisfies all the requirements of S .

8.1.2 Formalizing the Representation of Structure

A structural design representation was developed which was sufficient for developing a theory of slices (§4.1). The structural representation has the following characteristics:

- *Components* which are implementation artifacts that encapsulate behaviour and further structure.
- *Interaction Points* which define interfaces to components and possible interactions between components.
- *Component wirings* which define how the components are connected to interaction points.
- *Component decomposition* in order that components can be decomposed into simpler subcomponents.

8.1.3 Formally Linking Structure and Behaviour

LOTOS was chosen as the language for representing behaviour. In order to represent both slice expressions and component expressions as LOTOS expressions, a number of issues are resolved:

- *Mapping interactions to LOTOS events.* Using the abstract data types of LOTOS, an interaction between components is encoded as a LOTOS event (§4.2.1.2).
- *The wiring problem.* LOTOS cannot in general represent the arbitrary interconnection of components which interact by means of synchronous rendezvous. This research developed a method of interconnecting arbitrary networks of processes using the LOTOS operators (§4.2.1.2).

- *Component decomposition.* One problem this research identified involved comparing behaviour specifications which were developed at different levels of the structural hierarchy. Slice expressions contain component names within the events; when component decomposition occurs the subcomponents have different names from the components. To overcome the problem, a *structure renaming function* Δ is defined which maps events at one structural level to a higher structural level (§4.2.3). The structure renaming function can be automatically generated from the structure hierarchy tree.

8.1.4 Visual Methods for Design Capture

It is often difficult and awkward for design engineers to work with formal methods. To facilitate design capture in a formal representation a number of visual notations were developed.

The notation for structural design was defined in §4.1.1.1. The notation allowed structural design to be almost completely represented visually. Translation to the formal representation can be automated within a tool.

Behavioural information, particularly that involving a number of components, is very difficult for a designer to understand. In order to facilitate the capture and understanding of behavioural information three different methods of visualizing behaviour are developed:

- For an informal understanding of sequential slice behaviour, event sequences are superimposed directly upon the structural diagram.
- For representing sequences of interactions involving multiple components a *timeline* notation is developed (§4.2.2). The timeline notation allows the definition of parameterized processes and variables. A limited set of operators were also defined which were found to be sufficient for refining timelines by slice extension.

- A state transition notation is developed (§4.2.2) and used primarily to represent constraints on individual components and synchronizing processes between concurrent slice expressions.

Both the timeline and state transition notations can be translated into LOTOS expressions. A tool was developed which allows for the graphical entry of state transition diagrams and the automatic translation into a LOTOS process [57].

8.2 Conclusions

8.2.1 Applicability of Slices to an Effective Design Process

Ways in which slices can be useful during the design process are as follows:

- Slices provide an intuitive means for quickly specifying and experimenting with the behaviour of concurrent components.
- Given a slice S , the behaviour of component c which is implied by S can be determined syntactically or semantically. Once the behaviour of c implied by S has been determined, the designer can use this behaviour to assist in the development of the component specification B_c , or as a means of verifying that B_c satisfies all the requirements imposed by slice S .
- Visual notations can assist a designer in capturing and refining slices.
- Automated tools can be effectively applied in order to verify the *ext* relation, extract component behaviour from slice expressions, and analyze a slice for critical races.

Difficulties encountered when using slices include the following:

- Without tool support, the visual notations for specifying and refining slices are often difficult to use.
- Some of the verification and analysis techniques based on slices are intractable if a complete reachability analysis is performed.

- The behaviour specification of component c which can be extracted from slice S is not always an accurate representation of the behaviour of c .

8.2.2 LOTOS as a Behavioural Design Language

The LOTOS specification language was found to have a number of strengths when applied to the problem of formalizing slice expressions:

- *Different LOTOS styles.* The different styles for structuring a LOTOS specification are well suited to capturing the behaviour at different stages of the design process.
- *Process algebra for combining slices.* The LOTOS behaviour expression operators were found to be very useful for refining slice expressions. In particular the behaviour expression operators allowed timeline segments to be constructed into extensions of simpler slices and slice expressions to be combined in parallel with constraints in order to synchronize the concurrent behaviour of the slices.
- *Executable.* Since LOTOS is an executable language, a great deal of analysis and verification on slice expressions can be performed by executing the specification.

A number of difficulties were also encountered while using LOTOS:

- *Inefficient execution.* Current tools were limited in the size of resource oriented specifications which could be efficiently executed (§7.3). This appeared to be a result of having to use the *default* processes in order to overcome the wiring problem of LOTOS (§4.2.1.2).
- *Lack of modularity.* The way LOTOS is currently defined, it is not possible to have separately defined modules as would be found in most modern programming languages.
- *Interleaving model of concurrency.* LOTOS uses an interleaved model of concurrency, where an execution of the system is represented as a total

ordering of events. When combining slices in parallel, this exacerbates the problem of exponential explosion in the number of traces which must be analyzed for many verifications.

- *Lack of Structural Design.* LOTOS does not have any explicit means of representing design structure. This problem was overcome by encoding structural information within the abstract data types of LOTOS. However this did result in LOTOS specifications in which the structural information was often difficult for the designer to understand simply by looking at the LOTOS code.

8.2.3 Visualizing Structural and Behavioural Design

Three behaviour visualizations were used in this research:

- *Superimposing slices directly upon structure diagrams.* For simple slice expressions, superimposing the slice directly on the structure was an effective means of capturing behaviour.
- *Timelines.* Timelines were useful for capturing slice behaviour which was primarily sequential in nature. With tool support, they can be used to assist in the refinement of slices.
- *State transition diagrams.* This research used state transition diagrams to represent synchronization constraints between concurrent slices. They were found to be very helpful in representing such constraints.

8.3 Future Research

This thesis has provided a number of contributions to the formalization of a slice based design technique which can be applied to a process for designing concurrent systems. In order to integrate such an approach into a design environment, a number of areas require further research.

8.3.1 Tool Support

This research has laid down a theoretical foundation for using a slice based approach within a design process for concurrent systems, and one example of a tool was developed. Although some areas were identified where tools could be used to support the design process (Chapter 7), significant work is required to develop a full tool set. In developing a tool set, specific research topics which must be addressed are:

- *Efficient Techniques for Verification and Analysis.* A number techniques for analysis and verification were identified. Efficient and practical algorithms are required in order to perform these verifications in cases where the state space size is intractable or infinite. Areas of possible research include using a partial order semantics for behavioural models, heuristics for directing searches through the state space, or better methods for abstracting state information.
- *LOTOS Interpreters.* As identified in §7.3, there were performance problems with the LOTOS interpreter when applied to resource oriented specifications. Further research is needed to develop LOTOS interpreters which can efficiently interpret such specifications. This issue is critical if verification is to be done by reachability analysis; the larger the number of states which can be analyzed in a given amount of time, the more reliable the results of the analysis will be.

8.3.2 Using Slices

This research has illustrated how slices can be formalized and used during the design process. In order to confirm that these results are applicable to large scale systems, more case studies on using the slice style must be done in order to obtain empirical observations regarding the technique. Questions which must be answered include the following:

- What techniques can be used to refine slices? Are there standard methods and techniques for refining slices which maintain the *ext* relation?
- How can slices be applied to structural representations which are more expressive than the representation of this thesis?

8.3.3 Extensions to Structural Representation

The structural design representation of Chapter 4, although sufficient for developing the slice style of this research, is too simple to be used in an industrial setting. Further research is required to extend this representation. Possible areas for extension include:

- *Dynamic Structure.* Concurrent systems often are dynamic, with components and connections being created and deleted over time. Further research is required to extend the design representation to dynamic structure and integrate this with the behavioural representation.
- *Intercomponent Communication.* The structural design of this research limited component interactions to multiway rendezvous. Further research is required to extend this model to include other interaction types, such as asynchronous, broadcast, transient, etc., or to allow component interactions to include more complex, layered communication protocols.
- *Component Types.* This research restricted itself to looking at active components. A more complete design environment may require a more extensive set of component types, including active components, passive components, reactive components, monitors, etc.

Appendix A: Summary of Notation and Conventions

A'	Complement of set A.
$A \cap B$	Set intersection.
$A \cup B$	Set union.
$A - B$	Set difference: $B' \cap A$
$ A $	Number of elements in set A.
$\mathcal{P}(A)$	Powerset of A (set of all subsets).
$\{x \in X \mid p(x)\}$	Set definition. The set of elements x of X such that $p(x)$.

Figure A.1 Set notation.

$\forall x \in X \bullet p(x)$	Universal quantification. For all elements x in X , $p(x)$.
$\exists x \in X \bullet p(x)$	Existential quantification. There exists an element x in X such that $p(x)$.

Figure A.2 Logical quantifiers.

λ	Empty sequence
$a_1.a_2 \dots a_n$	Sequence of elements a_1 through a_n
$a_1; a_2 \dots a_n$	Sequence of elements a_1 through a_n (alternate notation)
A^*	The set of all finite length sequences of elements of A .
$t \upharpoonright A$	Sequence t restricted to elements of set A .

Figure A.3 Sequences.

$[]$	Empty list.
$[a_1, a_2, \dots, a_n]$	List of elements a_1 through a_n

Figure A.4 Lists.

$B \xrightarrow{e_1 \dots e_n} B'$	$\exists B_j, 1 \leq j < n \bullet B \xrightarrow{e_1} B_1 \xrightarrow{e_2} \dots \xrightarrow{e_n} B'$
$B \xrightarrow{e_1 \dots e_n} B', e_j \neq i$	$\exists B_j, 1 \leq j < 2n, k_l, 1 \leq l \leq n+1 \bullet$ $B \xrightarrow{i^{k_1}} B_1 \xrightarrow{e_1} B_2 \xrightarrow{i^{k_2}} \dots \xrightarrow{e_n} B_{2n} \xrightarrow{i^{k_{n+1}}} B'$
$traces(B)$	$\{t \in I^* \mid \exists B' \bullet B \xrightarrow{t} B'\}$
$after(B, t)$	$\{B' \mid B \xrightarrow{t} B'\}$
$refusals(B)$	$\{A \in \mathcal{P}(I) \mid$ $\exists B' \in after(B, \lambda) \bullet \forall e \in A \bullet e \notin traces(B')\}$
$failures(B)$	$\{\langle t, A \rangle \in tr(B) \times \mathcal{P}(I) \mid$ $\exists B' \in after(B, t) \bullet A \in refusals(B')\}$
$B_2 \text{ ext } B_1$	$traces(B_2) \supseteq traces(B_1)$ and $\forall t \in traces(B_1) \bullet \forall A \in \mathcal{P}(I) \bullet$ if $\langle t, A \rangle \in failures(B_2)$ then $\langle t, A \rangle \in failures(B_1)$

Figure A.5 Definitions of properties of transition system B .

<p>Given: structure $\langle IP, C \rangle$ interaction e component $c \in C$ component set $D \subseteq C$</p>	
I	Set of interactions of $\langle IP, C \rangle$.
$I(c)$	Set of interactions of $\langle IP, C \rangle$ in which component c is involved.
$I(D)$	Set of interactions of $\langle IP, C \rangle$ involving only components of D .
$C(e)$	Set of components of C involved in interaction e

Figure A.6 Interaction sets.

Appendix B: Summary of LOTOS Operators

Operator	Name	Example	Transition System
;	action prefix	a;exit	
[]	choice	(a;exit)[](b;exit)	
	interleaving	(a;exit) (b;exit)	
	full synchronization	((a;exit)[](b;exit)) ((a;exit)[](c;exit))	
G	synchronize on gates G	((a;exit)[](b;exit)) [a] ((a;exit)[](c;exit))	
>>	enable	a;exit >> b;exit	
[>	disable	a;exit [> b;exit	

Appendix C: Proof of Theorems

This appendix provides the proofs for the theorems of Chapter 5. The following conventions will be used in the proofs:

- S represents a LOTOS slice expression.
- B_c is a LOTOS component expression for component c .
- $B_{\langle IP, C \rangle}$ is the LOTOS resource oriented expression constructed by composing the component expressions $B_c, c \in C$ according to the method of §4.2.1.2.
- $\text{phi}_c(S)$ is the LOTOS component expression for component c which is derived from S by applying the phi transform described in §5.2.1.1 and Appendix E.

Lemma: 1.

For all interactions a , components c involved in interaction a , and LOTOS slice expressions S :

if
 $S \xrightarrow{a} S'$
then
 $\text{phi}_c(S) \xrightarrow{a} \text{phi}_c(S')$

Proof of lemma 1.

Given a LOTOS specification, the corresponding transition system can be constructed from the axioms and inference rules of transition defined as part of the LOTOS standard. These axioms and rules of transition can be used to derive the state transitions which are part of the behaviour tree: i.e., $S \xrightarrow{a} S'$ if and only if there is a finite length derivation based on the axioms and inference rules of transition which ‘prove’ that $S \xrightarrow{a} S'$. The lemma can be proved by showing that if $S \xrightarrow{a} S'$ by derivation D then there exists a derivation D' to prove that $\text{phi}_c(S) \xrightarrow{a} \text{phi}_c(S')$. The derivation D' can be constructed from derivation D by showing that all the axioms of transition conform to the lemma, and all the inference rules of transition preserve the

property of the lemma. Thus any derivation D of the transition $S \xrightarrow{a} S'$ will imply that $\text{phi}_c(S) \xrightarrow{a} \text{phi}_c(S')$ by derivation D' .

The derivation D' can be shown to exist by proving that:

- If $B \xrightarrow{a} B'$ is an axiom of transition, then $\text{phi}_c(B) \xrightarrow{a} \text{phi}_c(B')$; and
- If
 1. $\frac{\text{conditions}}{S \xrightarrow{a} S'}$ is an inference rule; and
 2. for all transitions $B \xrightarrow{e} B'$ which are part of the *conditions*,
 $\text{phi}_c(B) \xrightarrow{e} \text{phi}_c(B')$
 then $\text{phi}_c(S) \xrightarrow{a} \text{phi}_c(S')$.

The remainder of this proof looks at each axiom and inference rule in turn and proves that it preserves the required property. The following conventions are used in the proof:

- g represents a gate name.
- i is the LOTOS hidden event.
- d_k is an expression of the form ‘!expression’ or ‘?declaration’.
- if B is a behaviour expression and $[sub]$ is a substitution of actual gate names or data values for formal gate names or parameters then $[sub]B$ is the expression resulting from performing the substitution on expression B . Note that $\text{phi}_c([sub]B) = [sub]\text{phi}_c(B)$.

Axiom: $i; B \xrightarrow{i} B$

$$\begin{aligned} \text{phi}_c(i; B) &= i; \text{phi}_c(B) \text{ by definition of } \text{phi}_c \\ i; \text{phi}_c(B) &\xrightarrow{i} \text{phi}_c(B) \text{ by axiom of transition} \\ \therefore \text{phi}_c(i; B) &\xrightarrow{i} \text{phi}_c(B) \end{aligned}$$

Axiom: $gd_1 \dots d_n; B \xrightarrow{g^{v_1 \dots v_n}} [substitution]B$

$$\begin{aligned} \text{phi}_c(gd_1 \dots d_n; B) &= gd_1 \dots d_n; \text{phi}_c(B) \text{ by definition of } \text{phi}_c \\ gd_1 \dots d_n; \text{phi}_c(B) &\xrightarrow{g^{v_1 \dots v_n}} [substitution]\text{phi}_c(B) \\ &\text{by axiom of transition} \\ \therefore \text{phi}_c(gd_1 \dots d_n; B) &\xrightarrow{g^{v_1 \dots v_n}} \text{phi}_c([substitution]B) \end{aligned}$$

Axiom: $gd_1 \dots d_n[P]; B \xrightarrow{gv_1 \dots v_n} [substitution]B$

This axiom is similar to the previous axiom, however it contains a predicate $[P]$ to be evaluated. The transform phi_c is defined such that:

- If all the variables of P are defined then the predicate P is retained by the phi_c transform.
- If there are variables in P which are not defined then the predicate P is removed by the transform.

In either case, the proof is similar to the previous axiom.

Axiom: $\text{exit} \xrightarrow{\delta} \text{stop}$
 $\text{phi}_c(\text{exit}) = \text{exit}$
 $\text{phi}_c(\text{stop}) = \text{stop}$
 $\therefore \text{phi}_c(\text{exit}) \xrightarrow{\delta} \text{phi}_c(\text{stop})$

Axiom: $\text{exit}(E_1 \dots E_n) \xrightarrow{\delta v_1 \dots v_n} \text{stop}$

Proof is similar to previous axiom.

Inference rule: $\frac{[t/x_1 \dots t/x_n]B \xrightarrow{a} B'}{\text{let } x_1 = t_1 \dots x_n = t_n \text{ in } B \xrightarrow{a} B'}$
 $[t_1/x_1 \dots t_n/x_n]\text{phi}_c(B) \xrightarrow{a} \text{phi}_c(B')$
 $\text{let } x_1 = t_1 \dots x_n = t_n \text{ in } \text{phi}_c(B) \xrightarrow{a} \text{phi}_c(B)$ by inference rule
 $[\text{phi}_c(\text{let } x_1 = t_1 \dots x_n = t_n \text{ in } B)] =$
 $[\text{let } x_1 = t_1 \dots x_n = t_n \text{ in } \text{phi}_c(B)]$ by definition of phi_c
 $\therefore \text{phi}_c(\text{let } x_1 = t_1 \dots x_n = t_n \text{ in } B) \xrightarrow{a} \text{phi}_c(B')$

Inference rule: $\frac{B[g_i/g] \xrightarrow{a} B', 1 \leq i \leq n}{\text{choice } g \text{ in } [g_1 \dots g_n][B] \xrightarrow{a} B'}$
 $\text{phi}_c(B)[g_i/g] \xrightarrow{a} \text{phi}_c(B')$
 $\text{choice } g \text{ in } [g_1 \dots g_n][\text{phi}_c(B) \xrightarrow{a} \text{phi}_c(B')]$ by inference rule
 $\text{phi}_c(\text{choice } g \text{ in } [g_1 \dots g_n][B]) = \text{choice } g \text{ in } [g_1 \dots g_n][\text{phi}_c(B)]$
by definition of transform
 $\therefore \text{phi}_c(\text{choice } g \text{ in } [g_1 \dots g_n][B]) \xrightarrow{a} \text{phi}_c(B')$

$$\text{Inference rule: } \frac{[t/x]B \xrightarrow{\alpha} B'}{\text{choice } x \square B \xrightarrow{\alpha} B'}$$

Proof is similar to previous inference rule.

$$\text{Inference rules: } \frac{B[g_1/g] \text{op} \dots \text{op} B[g_n/g] \xrightarrow{\alpha} B'}{\text{par } g \text{ in } [g_1 \dots g_n] \text{op } B \xrightarrow{\alpha} B'}$$

Proof is similar to previous inference rule.

$$\text{Inference rules: } \frac{B \xrightarrow{\alpha} B', \text{name}(a) \notin \{g_1 \dots g_n\}}{\text{hide } g_1 \dots g_n \text{ in } B \xrightarrow{\alpha} B'} \quad \frac{B \xrightarrow{\alpha} B', \text{name}(a) \in \{g_1 \dots g_n\}}{\text{hide } g_1 \dots g_n \text{ in } B \xrightarrow{i} B'}$$

$$\begin{aligned} & \text{phi}_c(B) \xrightarrow{\alpha} \text{phi}_c(B') \\ & \text{hide } g_1 \dots g_n \text{ in } \text{phi}_c(B) \xrightarrow{\alpha} \text{phi}_c(B') \text{ or} \\ & \text{hide } g_1 \dots g_n \text{ in } \text{phi}_c(B) \xrightarrow{i} \text{phi}_c(B') \end{aligned}$$

In each case:

$$\begin{aligned} & \text{phi}_c(\text{hide } g_1 \dots g_n \text{ in } B) \xrightarrow{\alpha} \text{phi}_c(B') \text{ or} \\ & \text{phi}_c(\text{hide } g_1 \dots g_n \text{ in } B) \xrightarrow{i} \text{phi}_c(B') \end{aligned}$$

$$\text{Inference rule: } \frac{B \xrightarrow{\alpha} B', \text{name}(a) \neq \delta}{B \gg \text{accept } x_1, \dots, x_n \text{ in } B_2 \xrightarrow{\alpha} B' \gg \text{accept } x_1, \dots, x_n \text{ in } B_2}$$

$$\begin{aligned} & \text{phi}_c(B) \xrightarrow{\alpha} \text{phi}_c(B'), \text{name}(a) \neq \delta \\ & \text{phi}_c(B) \gg \text{accept } x_1, \dots, x_n \text{ in } \text{phi}_c(B_2) \xrightarrow{\alpha} \\ & \text{phi}_c(B') \gg \text{accept } x_1, \dots, x_n \text{ in } \text{phi}_c(B_2) \text{ by inference rule} \\ & \text{phi}_c(B \gg \text{accept } x_1 \dots x_n \text{ in } B_2) = \\ & \text{phi}_c(B) \gg \text{accept } x_1 \dots x_n \text{ in } \text{phi}_c(B_2) \\ & \text{by definition of transform} \end{aligned}$$

$$\begin{aligned} \therefore \text{phi}_c(B \gg \text{accept } x_1, \dots, x_n \text{ in } B_2) & \xrightarrow{\alpha} \\ \text{phi}_c(B' \gg \text{accept } x_1, \dots, x_n \text{ in } B_2) & \end{aligned}$$

Inference rules:

$$\frac{B \xrightarrow{\delta v_1 \dots v_n} B'}{B \gg \text{accept } x_1 \dots x_n \text{ in } B_1 \xrightarrow{i} [t_1/x_1 \dots t_n/x_n] B'}$$

$$\frac{B \xrightarrow{\alpha} B', \text{name}(a) \neq \delta}{B \gg B_1 \xrightarrow{\alpha} B' \gg B_1}$$

$$\frac{B \xrightarrow{\delta v_1 \dots v_n} B'}{B \gg B_1 \xrightarrow{\delta v_1 \dots v_n} B'}$$

$$\frac{B_1 \xrightarrow{\alpha} B'_1}{B[> B_1 \xrightarrow{\alpha} B'_1]}$$

Proof is similar to previous inference rule.

$$\text{Inference rule: } \frac{B \xrightarrow{\alpha} B', \text{ name}(a) \notin E}{B|E|B_1 \xrightarrow{\alpha} B'|E|B_1}$$

$$\begin{aligned} & \text{phi}_c(B) \xrightarrow{\alpha} \text{phi}_c(B') \\ & \text{phi}_c(B)|E|\text{phi}_c(B_1) \xrightarrow{\alpha} \text{phi}_c(B')|E|\text{phi}_c(B_1) \text{ by inference rule} \\ & \text{phi}_c(X|E|Y) = \text{phi}_c(X)|E|\text{phi}_c(Y) \text{ by definition of transform} \\ \therefore & \text{phi}_c(B|E|B_1) \xrightarrow{\alpha} \text{phi}_c(B'|E|B_1) \end{aligned}$$

Inference rules:

$$\frac{B \xrightarrow{\alpha} B', \text{ name}(a) \notin E}{B_1|E|B \xrightarrow{\alpha} B_1|E|B'}$$

$$\frac{B_1 \xrightarrow{\alpha} B'_1, B_2 \xrightarrow{\alpha} B'_2, \text{ name}(a) \in E'}{B_1|E|B_2 \xrightarrow{\alpha} B'_1|E|B'_2}$$

$$\frac{B \xrightarrow{\alpha} B'}{B[] B_1 \xrightarrow{\alpha} B'}$$

$$\frac{B \xrightarrow{\alpha} B'}{B_1[] B \xrightarrow{\alpha} B'}$$

Proof is similar to previous inference rule.

Inference rules:

$$\frac{B_1[] [] B_2 \xrightarrow{\alpha} B'}{B_1[] [] B_2 \xrightarrow{\alpha} B'}$$

$$\frac{B_1|[g_1 \dots g_n]| B_2 \xrightarrow{\alpha} B', \{g_1 \dots g_n\} = G}{B_1[] B_2 \xrightarrow{\alpha} B'}$$

These are specialized cases of the general parallel operator and the proof is similar.

Inference rule: $\frac{B \xrightarrow{a} B', D \vdash P}{[P] - > B \xrightarrow{a} B'}$

$\text{phi}_c(B) \xrightarrow{a} \text{phi}_c(B')$
 $[P] - > \text{phi}_c(B) \xrightarrow{a} \text{phi}_c(B)$ by inference rule.

Either $\text{phi}_c([P] - > B) = [P] - > \text{phi}_c(B)$
 or $\text{phi}_c([P] - > B) = \text{phi}_c(B)$ by definition of transform.

In either case $\text{phi}_c([P] - > B) \xrightarrow{a} \text{phi}_c(B')$

Inference rule: $\frac{([t_1/x_1 \dots t_m/x_m]B_p)[g_1/h_1 \dots g_n/h_n] \xrightarrow{a} B'}{p[g_1 \dots g_n](t_1 \dots t_m) \xrightarrow{a} B'}$

Where process p is defined as $p[g_1 \dots g_n](t_1 \dots t_m) := B_p$

$([t_1/x_1 \dots t_m/x_m]\text{phi}_c(B_p))[g_1/h_1 \dots g_n/h_n] \xrightarrow{a} \text{phi}_c(B')$
 $\text{phi}_c(p[g_1 \dots g_n](t_1 \dots t_m)) = p'[g_1 \dots g_n](t_1 \dots t_m)$
 where $p'[g_1 \dots g_n](t_1 \dots t_m) := \text{phi}_c(B_p)$.
 $\therefore \text{phi}_c(p[g_1 \dots g_n](t_1 \dots t_m)) \xrightarrow{a} \text{phi}_c(B')$

■

Lemma: 2.

For all interactions a , components c not involved in interaction a , and LOTOS slice expressions S :

if

$$S \xrightarrow{a} S'$$

then

$$phi_c(S) = phi_c(S')$$

Proof of lemma 2.

Lemma 2 is similar to lemma 1 but deals with the case where the event causing the transition does not represent an interaction involving component c . The proof is similar to the proof of lemma 1: first by showing that the lemma is true for the axioms of transition; and then showing that if the lemma is true for all the transitions of the preconditions of an inference rule, then it is true for the transitions in the consequence of the transition rule. The proof is not difficult and due to its similarity with the previous lemma will not be presented here. ■

Lemma: 3.

If

S_{de} is derived from S by applying a data extension transformation as defined in §5.2.1.3

then

$$S_{de} \text{ ext } S.$$

Proof of lemma 3.

It can easily be shown that $S_{de} \text{ ext } S$ by noting that a data extension consists of replacing a constant expression k in S by a variable v of the appropriate sort. By the semantics of LOTOS, a variable represents a choice between all the possible values of the sort; by letting S_{de} ‘choose’ v to be k , the expression S_{de} can behave like S . ■

Lemma: 4.

If

S' is derived from S by applying the simplifying transformations defined in the table of figure 5.11 (page 128)

then

$$failures(S) = failures(S').$$

Proof of lemma 4.

Proof that the simplifying transformations preserve failure equivalence can be easily performed by reference for example to [18,71,46]■

Lemma: 5.

$t \in traces(B_{\langle IP, C \rangle})$ if and only if $\forall c \in C \bullet t \upharpoonright I(c) \in traces(B_c)$

Proof of lemma 5.

Assume $t \in traces(B_{\langle IP, C \rangle})$. Therefore, by the way in which $B_{\langle IP, C \rangle}$ is constructed, $t \in traces(B_c \parallel\parallel default(c))$. Since $default(c)$ includes only those interactions in which component c is not involved, therefore $t \upharpoonright I(c) \in traces(B_c)$.

Assume $\forall c \in C \bullet t \upharpoonright I(c) \in traces(B_c)$. Therefore, $\forall c \in C \bullet t \in traces(B_c \parallel\parallel default(c))$ and it follows that $t \in traces(B_{\langle IP, C \rangle})$.■

Theorem: 1.

$$\forall t \in traces(S) \bullet t \upharpoonright I(c) \in traces(\phi_c(S))$$

Proof of theorem 1.

Using lemmas 1 and 2 it can be shown that $t \upharpoonright I(c) \in traces(phi_c(S))$.

The expression $\phi_c(S)$ is derived from $phi_c(S)$ by simplifying term rewrites and data extensions.

By lemmas 3 and 4 any trace of $phi_c(S)$ is also a trace of $\phi_c(S)$.

Therefore $t \upharpoonright I(c) \in traces(\phi(S))$. ■

Theorem: 2.

$$\forall t \in \text{traces}(S) \bullet t \upharpoonright I(c) \in \text{traces}(B_c)$$

Proof of theorem 2.

Let t be a *trace* of S .

By assumption 2 (page 116) $t \in \text{traces}(B_{\langle IP, C \rangle})$.

Therefore by lemma 5 $t \upharpoonright I(c) \in \text{traces}(B_c)$. ■

Theorem: 3.

For all components c , $t \in \text{traces}(S)$, $A_c \subseteq I(c)$

if

$$\langle t \upharpoonright I(c), A_c \rangle \in \text{failures}(B_c)$$

then

$$\forall u \in \{v \in \text{traces}(S) \mid v \upharpoonright I(c) = t \upharpoonright I(c)\} \bullet \langle u, A_c \rangle \in \text{failures}(S)$$

Proof of theorem 3.

Let t be a *trace* of S such that $\langle t \upharpoonright I(c), A_c \rangle \in \text{failures}(B_c)$.

Let u be any *trace* of S such that $u \upharpoonright I(c) = t \upharpoonright I(c)$. Since

$\langle u \upharpoonright I(c), A_c \rangle \in \text{failures}(B_c)$ it follows that $\langle u, A_c \rangle \in \text{failures}(B_{\langle IP, C \rangle})$.

But by assumption 2 (page 116) $B_{\langle IP, C \rangle} \text{ ext } S$ and therefore

$\langle u, A_c \rangle \in \text{failures}(S)$. ■.

Theorem: 4.

$$B_{\langle IP, C \rangle} \text{ ext } \phi(S)$$

Proof of theorem 4.

By assumption 3 (page 126), for all components c , $B_c \text{ ext } \phi_c(S)$. Since the component specification of c and $\text{default}(c)$ share no interactions in common it follows that:

$$[B_c \parallel \text{default}(c)] \text{ ext } [\phi_c(S) \parallel \text{default}(c)]$$

Noting that $(A_1 \text{ ext } A_2)$ and $(B_1 \text{ ext } B_2)$ implies:

$$A_1 \parallel B_1 \text{ ext } A_2 \parallel B_2$$

and by the way in which the resource oriented specifications are constructed from the component expressions and the `default` processes, it can be seen that $B_{\langle IP, C \rangle} \text{ ext } \phi(S)$. ■

Theorem: 5.

$$\text{traces}(\phi(S)) \supseteq \text{traces}(S)$$

Proof of theorem 5.

Let t be any trace of S .

By theorem 1, for any c , $t \upharpoonright I(c)$ is a trace of $\phi_c(S)$.

Therefore by lemma 5 t is a trace of $\phi(S)$. ■

Theorem: 6.

For all $D \subseteq C$, $t \in \text{traces}(S)$ and sets of interactions A_D involving only components of D :

if

$$\langle t \upharpoonright I(D), A_D \rangle \in \text{failures}(B_{\langle IP, D \rangle})$$

then

$$\langle t, A_D \rangle \in \text{failures}(S)$$

Proof of theorem 6.

Let t be a *trace* of S , D a set of components, and A_D a set of interactions such that $\langle t \upharpoonright I(D), A_D \rangle \in \text{failures}(B_{\langle IP, D \rangle})$. Since the set of components D can refuse the interactions A_D after engaging in trace $t \upharpoonright I(D)$, it follows that the set of components C can also refuse the interactions A_D after engaging in trace t . Therefore $\langle t, A_D \rangle \in \text{failures}(B_{\langle IP, C \rangle})$. But since $B_{\langle IP, C \rangle} \text{ ext } S$ therefore $\langle t, A_D \rangle \in \text{failures}(S)$. ■

Appendix D: Case Study – A Telephony System

This appendix investigates the use of slice expressions in describing the behaviour of a telephony system. The examples of this appendix are a summary of some of the work which can be found in the case study of [26].

The system which will be developed is a telephony example. A number of telephone handsets are connected to a central switching device. The handsets are used to establish voice communication between users of the system.

The design presented in this chapter is not intended as a representation of an actual telephony system. The objective of the case study is to explore specification styles involving numerous concurrent components through multiple levels of decomposition. As a result the component decomposition is artificial and intended only to illustrate the design principles involved rather than to represent a real design. In particular, structural decomposition is performed to a finer grained level of detail than is required to represent true system concurrency.

D.1 Problem Description

The problem to be addressed is the design of a private branch exchange (PBX) involving a number of phone sets attached to a central switch. External to the system there is a set of users, with a unique user associated with each phone set. A phone set communicates with its corresponding user in the following ways:

- A bell to notify the user of incoming calls.
- A speaker by which the user can hear signal tones or voice data. Signal tones notify the user of the status of a call, such as ‘ringing’, ‘busy’, etc.
- A microphone to accept voice data from the user.
- A keyboard by which the user can input numbers to the telephone.

Many of the LOTOS specifications of this case study were written in conjunction with Philippe Goldstein.

- A receiver hook by which the user can engage or hang up the phone.

The primary service provided by the system is Plain Old Telephone Service (POTS) in which a user picks up the receiver and dials a number. If the number dialed is ready to accept a call, the dialed telephone will ring; else a busy signal will be returned to the caller. If the called telephone rings and is answered then a call is established and the two participants can talk to each other until either participant hangs up the phone.

A call transfer service was also implemented whereby a user after receiving a call can transfer the call to a third party. The source and the new destination now have a call established while the original destination is no longer involved in the call. The discussion in this chapter is limited to the POTS service; a specification which includes the call transfer can be found in the more detailed description of the case study [26].

D.2 Structural Specification

The top level decomposition is given in figure D.1. It shows a set of external `user` components, a set of `phoneset` components, and a central `switch`. The users and phonesets communicate using the following interactions:

- `pickup`: The handset is picked up.
- `release`: The handset is replaced.
- `talk`: Voice data is input to microphone.
- `listen`: Sound is output on speaker. The sound can be either voice data or a signal such as `busy`, `ringing`, etc.
- `dial`: A number is dialed on the phone set. For modeling purposes the dialing of a complete number will be viewed as a single interaction.
- `ringing`: The bell on the phoneset is turned off or on.

The `switch` component is structurally decomposed as shown in figure D.2. The components contained within `switch` are:

- **`simplug(ID)`**. For each phoneset there is a corresponding component within the central switch called `simplug`. The identifier `ID` of `simplug` is the same as the identifier `ID` of the corresponding `phoneset`. The purpose of the component `simplug` is to record the state of the `phoneset` and to perform and service requests received from the phoneset.
- **`service`**. The `service` component is responsible for maintaining connections between phonesets. It maintains as part of its database a list of all connections and controls the state of the connections.
- **`internalLine(N)`**. `internalLine` is a pool of components which transfer voice data between telephone sets (it is assumed that specialized hardware is required for the transfer of voice data). There are a limited number of internal lines and a connection between phonesets cannot be established unless an internal line is available.
- **`allocator`**. The `allocator` component is a dispatcher, controlling access to the `internalLine` components. In order for a call to be established, the `service` must request an internal line from the `allocator`.

The only component within the `switch` which is decomposed is the `simplug` (figure D.3). There is one `simplug` for every phoneset in the system. The purpose of the `simplug` is to control and monitor the corresponding phoneset. The components into which it is divided are:

- The `decoder` and `encoder` which provide an interface to the phonesets.
- The `senderBuffer` and `receiverBuffer` which provide internal buffering between the `simplug` and the `internalLine` components.
- The component `pretreatment` which includes most of the ‘intelligence’ of `simplug`, recording the state information of the corresponding `phoneset`.

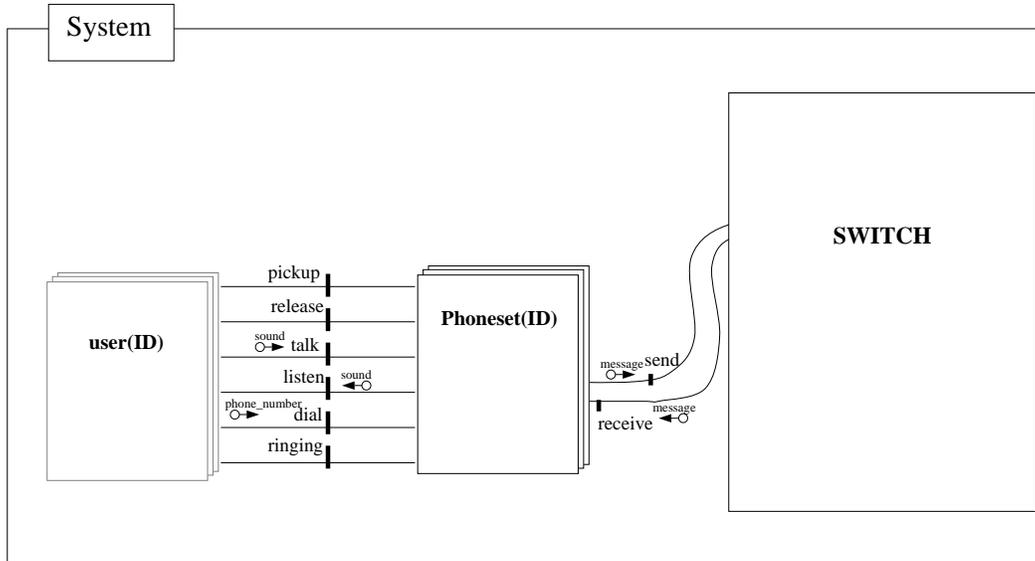


Figure D.1 Structure diagram for telephony example — phoneset and central switch.

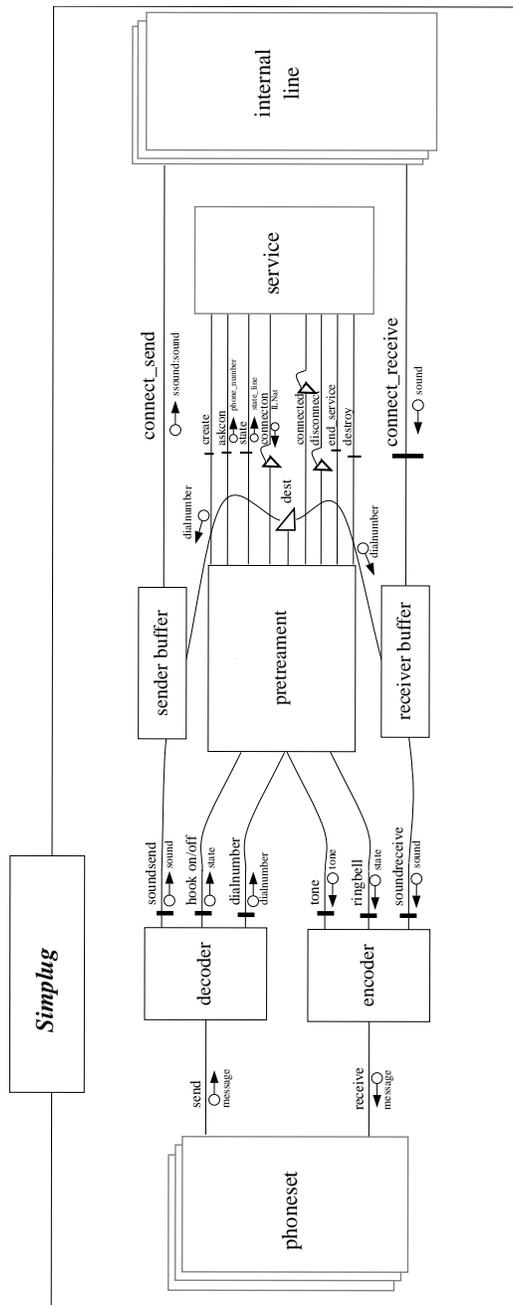


Figure D.3 Structure diagram for telephony example — **simplug** subcomponents.

D.3 Slice Specification

This section develops a number of slices for the telephony system. The slices are developed at different levels of the structural hierarchy. In particular, the following slices are developed:

- A simple slice showing a successful call completion is defined at the structural hierarchy levels `system`, `switch`, and `simplug`
- At the `switch` level, the slice representing call completion is extended to illustrate different outcomes, such as no internal line available, or busy signal returned.
- At the `switch` level, slices are combined to show concurrent calls between different users.

D.3.1 A Simple Slice — Successful Call Completion

This section shows three related slices at three different levels of the design hierarchy. The three slices all illustrate the expected behaviour during a successful call from user 1 to user 2. The slices will illustrate the behaviour of:

- the `system` level components.
- the `switch` level components.
- the `simplug` level components.

Figure D.4 defines process `linearSystem` which is a slice for the system level components showing a complete call between user 1 and user 2. The slice illustrates the behaviour when the call is completed successfully, voice data is transmitted, and the initiator of the call disconnects the call. The various stages of the call are the following (annotations are provided in the diagram corresponding to the following numbering):

1. The call source initiates the call by picking up the receiver and dialing the number.

2. The `switch` establishes the connection between the two phonesets.
3. Sound data is transferred from the initiator of the call to the recipient.
4. The initiator of the call terminates the call.

Figure D.5 defines the expression `linearSwitch` which illustrates the interaction sequence required of the `switch` level components in order to complete a call from `user(1)` to `user(2)`. Note that this expression has embedded within it all the interactions on the `switch` axis of expression `linearSystem` of figure D.4. The variable `n` represents the ID of the `internalLine` which is allocated to the call.

The stages of the call are the following:

1. The call is initiated by the source.
2. The `service` component verifies that the destination is not in use.
3. The `service` component gets an internal voice line from the `allocator`.
4. The connection is established between the `simplug` components and they notify the corresponding handsets.
5. Voice data is transmitted from the source to the destination.
6. A disconnection is initiated by the source.

The slice style of specification can be continued down the structural tree by providing a slice for the `simplug` components. Figure D.6 shows a slice for `simplug(1)` defined using the process `linearSimplug`. The slice illustrates the behaviour when `phoneset(1)` tries to establish a connection to `simplug(2)`. Note that this slice has embedded within it the interactions of the `simplug(1)` axis of the expression `linearSwitch`.

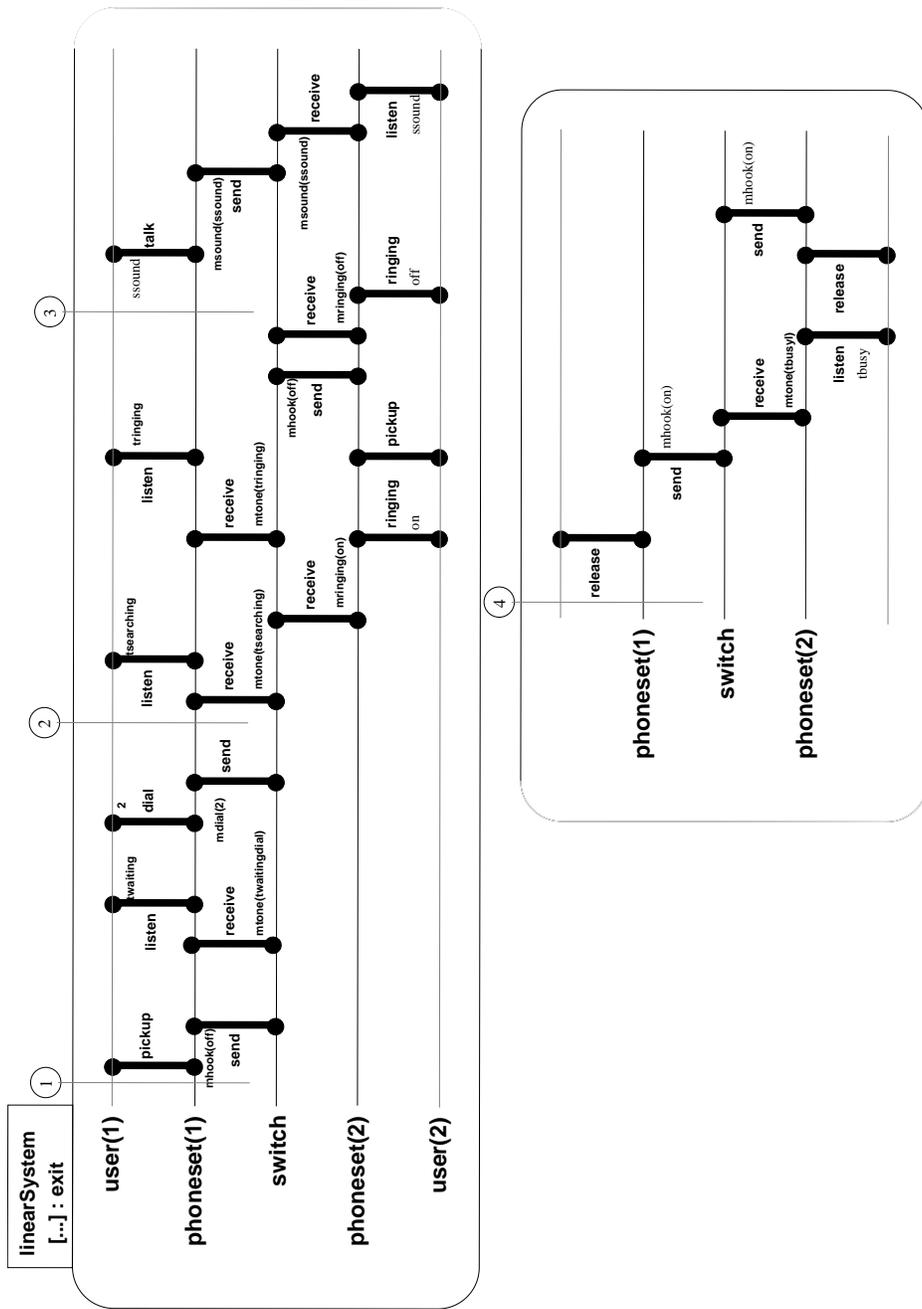


Figure D.4 Simple slice: system level components.

D.3.2 Extending Slices

Simple slices can be extended to show different possible outcomes from a set of triggering events at the edge of the system. This section will extend the simple slice of the `switch` level components to illustrate the following possible outcomes:

- User 1 may disconnect the call before it is completed.
- A busy signal is returned if no `internalLine` is available. This is represented by having a choice within the `switch`; when a line is requested the `allocator` will make a choice as to whether to grant access to a line or report no line available.
- The destination of the call is busy requiring a busy signal to be returned to user 1.
- After a call is completed either the user 1 or user 2 can terminate the call.

Extending a slice involves segmenting the slice, and then constructing a new slice from the segments. The slice expression `linearSwitch` defined in figure D.5 is segmented as illustrated in figures D.7, D.8 and D.9. Construction of these slices into an extended slice is done in figure D.10. The parameterized timelines segments defined in these figures are the following:

`initiate`. Initiate the call.

Parameters passed: `pn1, pn2 : phonesetID`

--IDs of phonesets involved in call.

Parameters returned: none.

`dial`. The source inputs the ID of the destination.

Parameters passed: `pn1, pn2 : phonesetID`

--IDs of phonesets involved in call.

Parameters returned: none.

`dialDisconnect`. The source hangs up before the service is notified of the destination ID.

Parameters passed: `pn1, pn2 : phonesetID`

--IDs of phonesets involved in call.

Parameters returned: none.

getService. Notify **service** of the intention to establish a connection.

Parameters passed: **pn1, pn2 : phonesetID**

--IDs of phonesets involved in call.

Parameters returned: **N : serviceID**

-- ID of internal service allocated.

busy. The destination is not available to receive the call.

Parameters passed: **pn1, pn2 : phonesetID**

--IDs of phonesets involved in call.

N : serviceID

-- ID of internal service allocated.

Parameters returned: none

notBusy. Determine that the destination is not busy.

Parameters passed: **pn1, pn2 : phonesetID**

--IDs of phonesets involved in call.

N : serviceID

-- ID of internal service allocated.

Parameters returned: **N : serviceID**

-- ID of internal service allocated.

openIL. Setup the **internalLine** for the call; establish the connection and notify the appropriate **simplug** components.

Parameters passed: **pn1, pn2 : phonesetID**

--IDs of phonesets involved in call.

N : serviceID

-- ID of internal service allocated.

Parameters returned: **N : serviceID**

-- ID of internal service allocated.

lineID : internalLineID

-- ID of internal line allocated.

noLine. No **internalLine** is available; disconnect.

Parameters passed: **pn1, pn2 : phonesetID**

--IDs of phonesets involved in call.

N : serviceID

-- ID of internal service allocated.

Parameters returned: none

accept. Destination accepts the call.

Parameters passed: **pn1, pn2 : phonesetID**

--IDs of phonesets involved in call.

N : serviceID

-- ID of internal service allocated.

lineID : internalLineID

-- ID of internal line allocated.

Parameters returned: **N : serviceID**

-- ID of internal service allocated.

lineID : internalLineID

-- ID of internal line allocated.

refuse. Destination refuses the call.

Parameters passed: **pn1, pn2 : phonesetID**

--IDs of phonesets involved in call.

N : serviceID
-- ID of internal service allocated.
lineID : internalLineID
-- ID of internal line allocated.

Parameters returned: **N : serviceID**
-- ID of internal service allocated.
lineID : internalLineID
-- ID of internal line allocated.

sendsound. Sound is transferred between users.

Parameters passed: **pn1,pn2 : phonesetID**
--IDs of phonesets involved in call.
N : serviceID
-- ID of internal service allocated.
lineID : internalLineID
-- ID of internal line allocated.

Parameters returned: **N : serviceID**
-- ID of internal service allocated.
lineID : internalLineID
-- ID of internal line allocated.

disconnect. The connection is terminated when one user hangs up.

Parameters passed: **pn1,pn2 : phonesetID**
--IDs of phonesets involved in call.

N : serviceID
-- ID of internal service allocated.
lineID : internalLineID
-- ID of internal line allocated.

Parameters returned: none

The timeline segments can be constructed into a slice expression representing the call connection including the various options; this is done in figure D.10 where the timelines are combined to define the process `branchSwitchProcess` which specifies a call between phonesets `pn1` and `pn2`. This process is then instantiated to give the specification of `branchSwitch`.

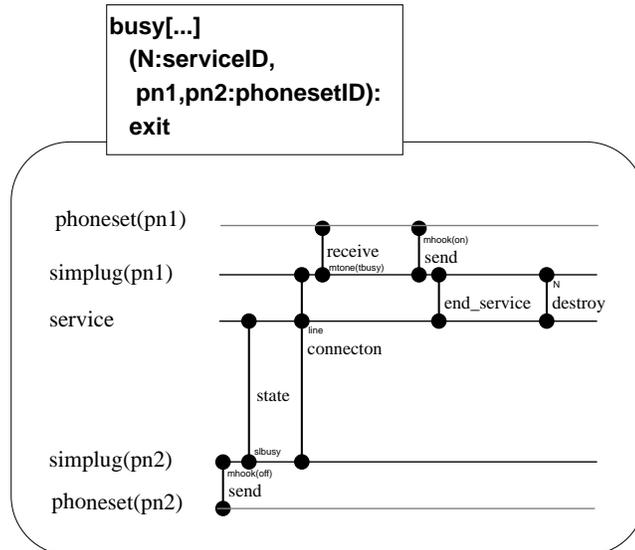
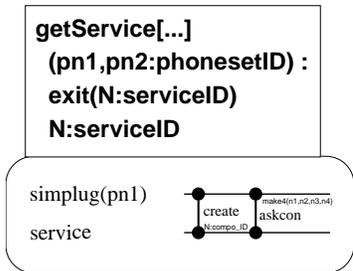
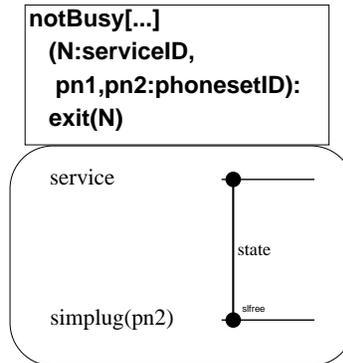
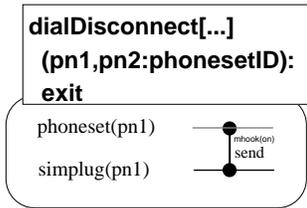
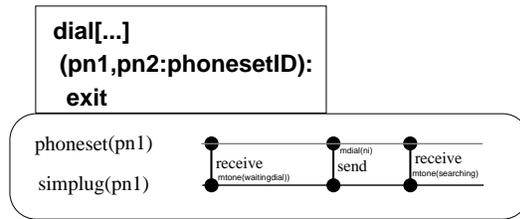
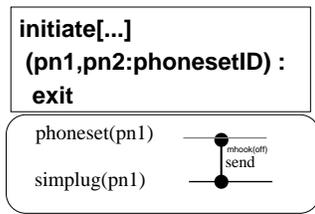
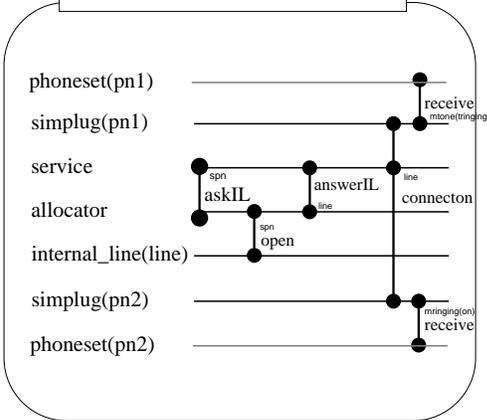
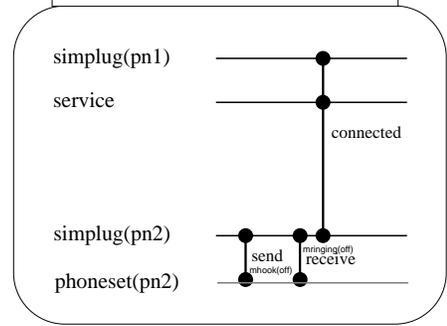


Figure D.7 Segmenting a slice: switch level components.

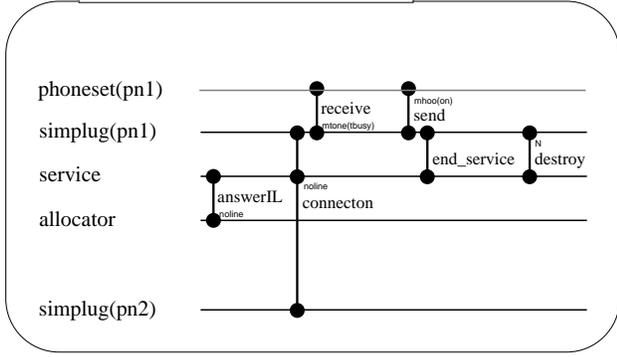
openIL[...]
 (N:serviceID,
 pn1,pn2:phonesetID,) :
 exit(N,line)
 line : lineID



accept[...]
 (N:serviceID,
 pn1,pn2:phonesetID,
 line:lineID) :
 exit(N,line)



noline[...]
 (N:serviceID,
 pn1,pn2:phonesetID,
 line:lineID) :
 exit



refuse[...]
 (N:serviceID,
 pn1,pn2:phonesetID,
 line:lineID) :
 exit

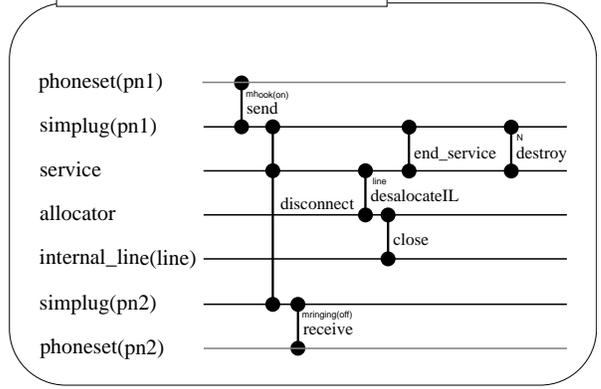


Figure D.8 Segmenting a slice: switch level components.

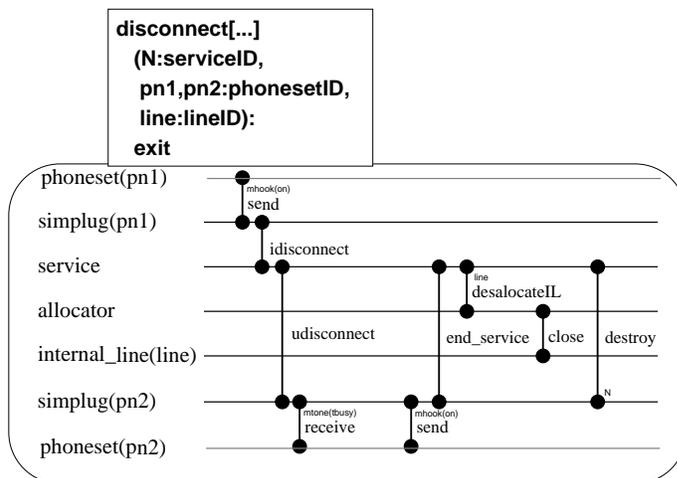
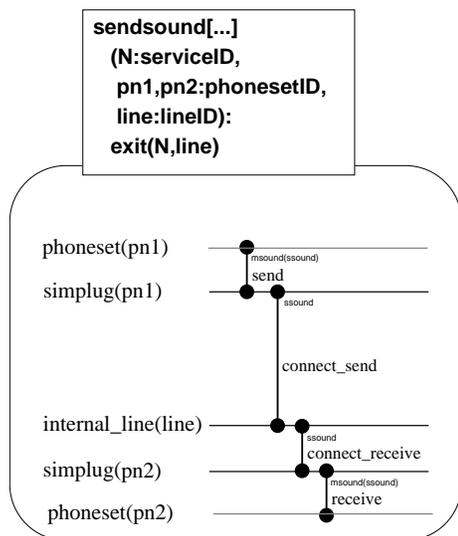


Figure D.9 Segmenting a slice: switch level components.

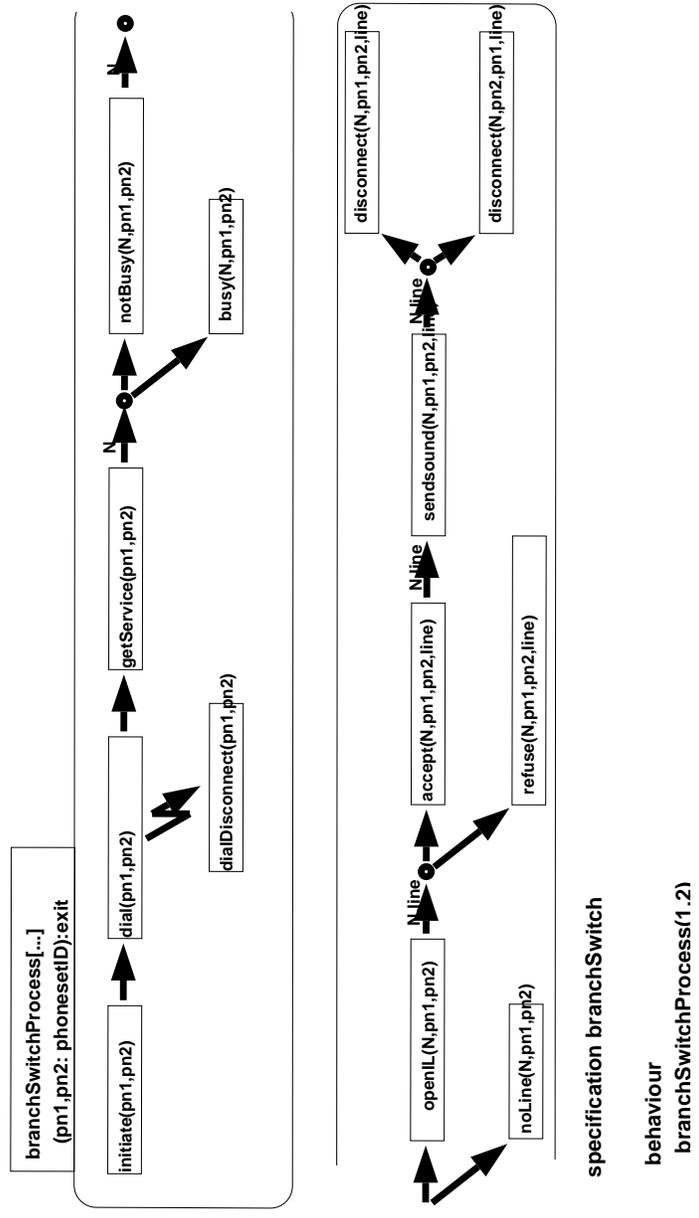


Figure D.10 Constructing a slice representing concurrency: **branchSwitch**.

D.3.3 Combining slices

This section combines slices to show three users, 1, 2, and 3, calling each other.

A process representing a call between two users was defined as `branchSwitch-Process` in figure D.10. This process can be instantiated with different data values to represent calls between different users. These processes can be combined in parallel in order to represent concurrent calls within the system.

Combining the processes in parallel requires that the designer identify the effect which concurrent slices have on each other, and then specify processes which synchronize the concurrent slice expressions (see §3.1.2 and §6.2).

In order to combine slices, interactions which affect other slices must be identified and removed. The only interaction shared between different concurrent slices is:

```
send!phoneset (ID) !simplug (ID)mhook (X)
```

which indicates whether a phoneset is busy, or whether it is willing to accept a call. Removing this interaction from `branchSwitch` is done by defining process `call` (figures D.11 and D.12).

The synchronizing processes used to combine the slices are the following:

- `onecall` (figure D.13) which constrains a phone set to being engaged in at most one call at a time.
- `busyconstraint1` (figure D.14) specifies whether a busy signal or a connect confirm is returned from a particular phoneset.
- `oneline` (figure D.15) which constrains an internal line to being engaged in at most one call at a time.
- `busylineconstraint` (figure D.16) which specifies that a busy signal is to be returned if no internal lines are available to handle the call.

The resulting LOTOS behaviour expression `concurrentConnect` is illustrated in figure D.17. The concurrent slice expression shows three users and two `inter-`

`naLine` components. Each process invocation `call` represents a call connection (with possible busy signal being returned) between two phone sets. The processes `onecallperphoneconstraint`, `busyconstraint`, `onecallperline`, and `busylineconstraint` are the synchronizing processes between the different concurrent slices.

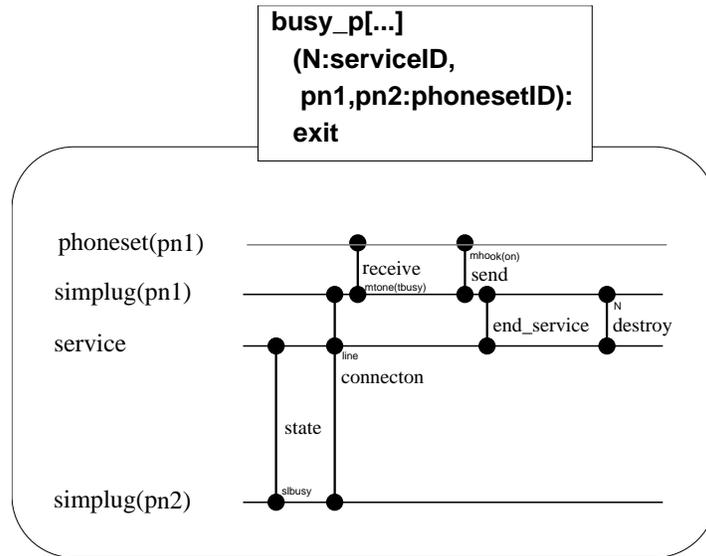


Figure D.11 Removing shared interactions **switch** level: defining process **busy_p** by removing interaction **send**.

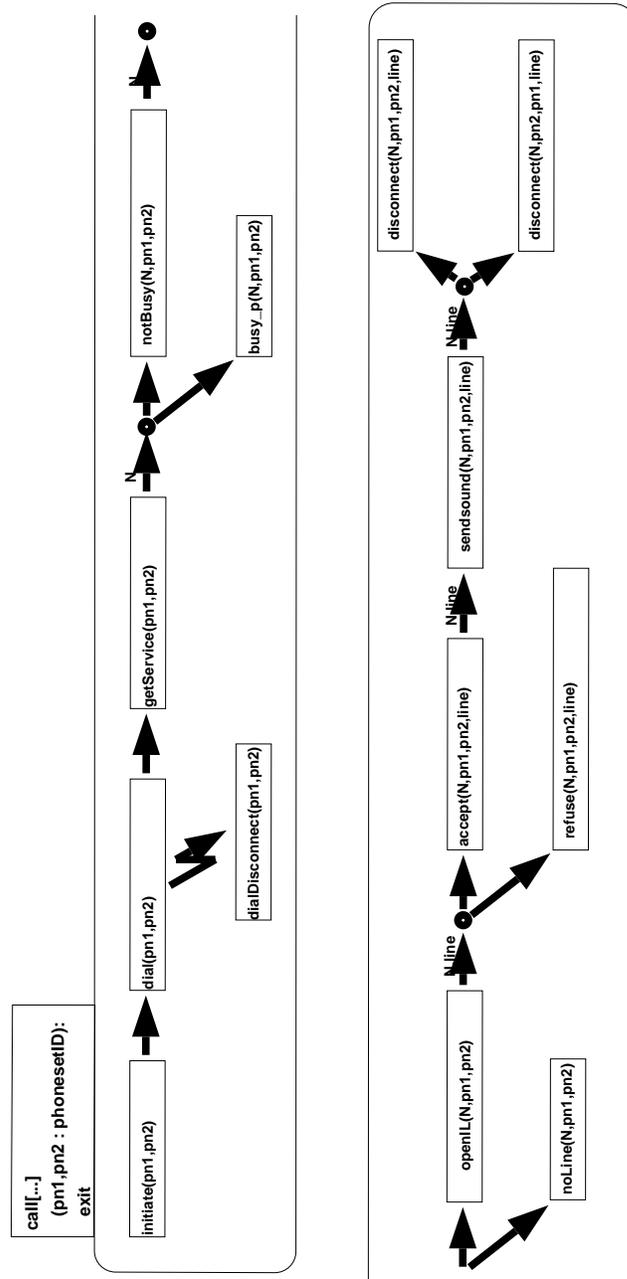


Figure D.12 Removing shared interactions **switch** level: a process defining connection establishment.

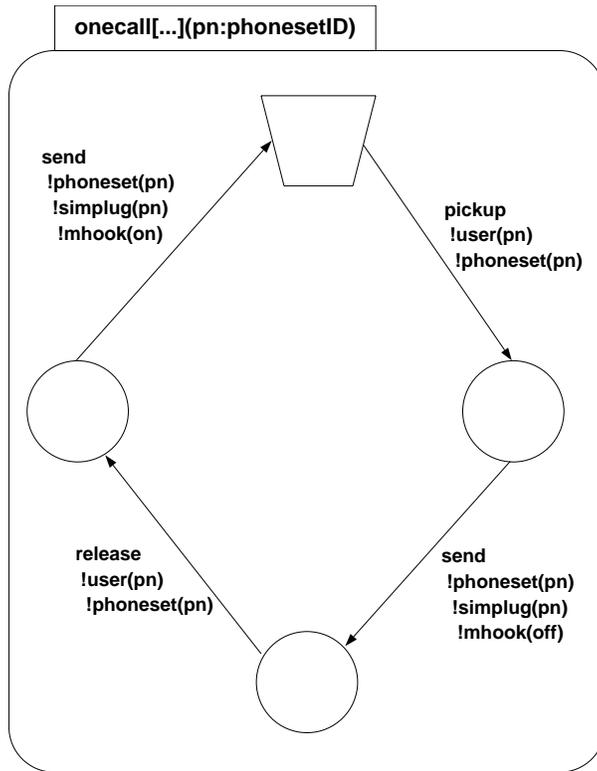


Figure D.13 Synchronizing process for `switch` level: single call per phoneset.

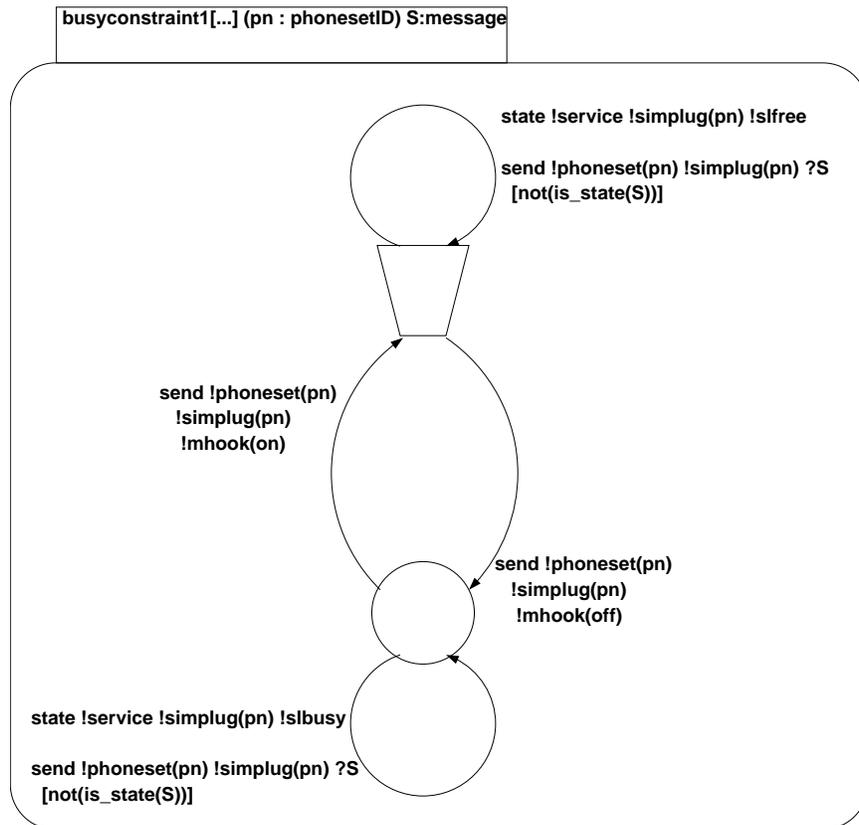


Figure D.14 Synchronizing process for **switch** level: busy signal returned if destination phone in use.

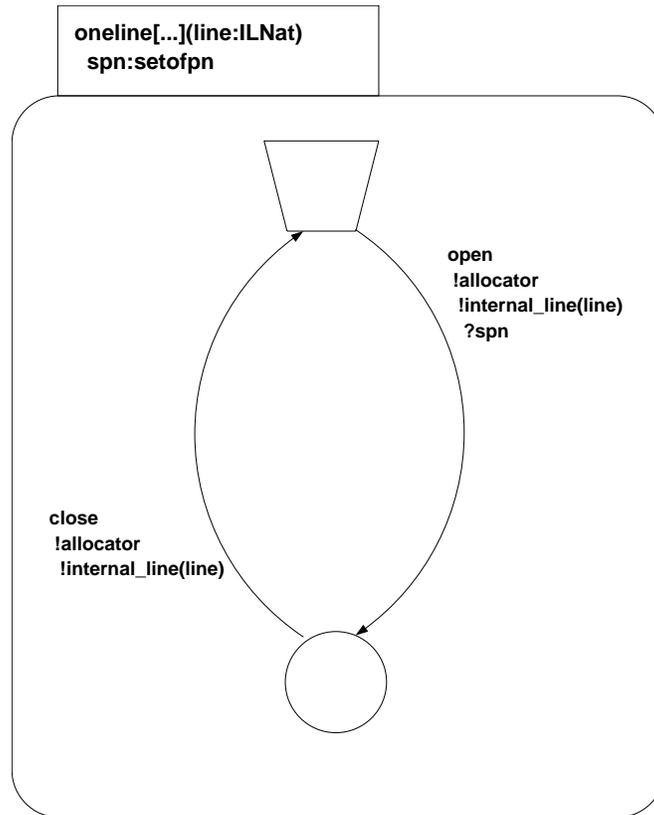


Figure D.15 Synchronizing process for **switch** level: single call per internal line.

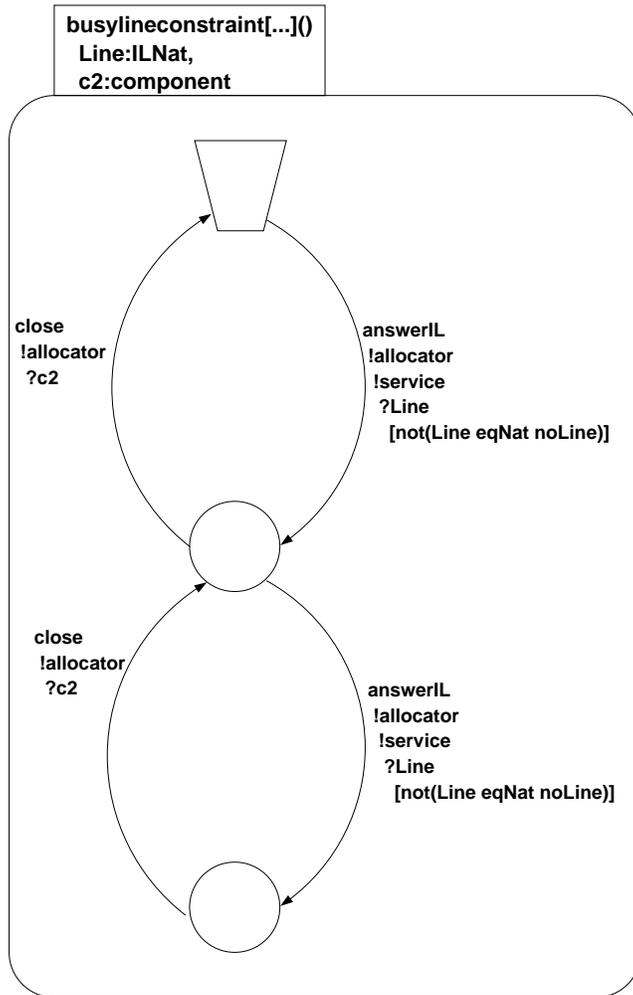


Figure D.16 Synchronizing process for **switch** level: busy signal returned if no **internalLine** component available.

```

(*****
*** POTS specification in LOTOS using the slice style ***
*** version 1.0 ***
*****
*)
(*****

specification concurrentConnect
  [pickup,dial,ringing,talk,listen,release,state,answerIL,open,close,send,
  receive,askcon,destroy,create,connected,disconnect,connecton,,
  connect_receiveconnect_send,end_service,askIL,desallocateIL]
  : noexit

(*****
*** Behaviour and process specification ***
*****

behaviour

(
(
(
(
call
  [pickup,dial,ringing,talk,listen,release,state,answerIL,open,
  close,send,receive,askcon,destroy,create,connected,disconnect,
  connecton,connect_receive,connect_send,end_service,askIL,desallocateIL]
  (make4(1,1,1,1),make4(2,2,2,2))
|||
call
  [pickup,dial,ringing,talk,listen,release,state,answerIL,open,
  close,send,receive,askcon,destroy,create,connected,disconnect,
  connecton,connect_receive,connect_send,end_service,askIL,desallocateIL]
  (make4(1,1,1,1),make4(3,3,3,3))
|||
call
  [pickup,dial,ringing,talk,listen,release,state,answerIL,open,
  close,send,receive,askcon,destroy,create,connected,disconnect,
  connecton,connect_receive,connect_send,end_service,askIL,desallocateIL]
  (make4(2,2,2,2),make4(3,3,3,3))
)
| [pickup,release,send] |
oncallperphoneconstraint [pickup,release,send]
| [send,state] |
busyconstraint [send,state]
| [open,close] |
oncallperline [open,close]
| [answerIL,close] |
busylineconstraint [answerIL,close]

where

(* SYNCHRONIZING PROCESS *)

process oncallperphoneconstraint [pickup,release,send]: noexit:=
  oncall [pickup,release,send] (make4(1,1,1,1))

```

Figure D.17 Combining slices **switch** level: behaviour **concurrentConnect** (Continued ...)

```

|||
onecall[pickup,release,send](make4(2,2,2,2))
|||
onecall[pickup,release,send](make4(3,3,3,3))
|||
onecall[pickup,release,send](make4(4,4,4,4))

where
  process onecall[pickup,release,send](pn:phone_number) : noexit :=
    service[pickup,release,send](begin,pn)

  where

    process service[pickup,release,send](s:stmstate,pn:phone_number): noexit :=

      [s eq begin] -> ( pickup!external!phoneset(pn);
        service[pickup,release,send](s1,pn) )
      []
      [s eq s1] -> ( send !phoneset(pn) !simplug(pn) !mhook(off);
        service[pickup,release,send](s2,pn) )
      []
      [s eq s2] -> ( release!external!phoneset(pn);
        service[pickup,release,send](s3,pn) )
      []
      [s eq s3] -> ( send !phoneset(pn) !simplug(pn) !mhook(on);
        service[pickup,release,send](begin,pn) )
      []
      (send !phoneset(pn) !simplug(pn) ?S:message [not(is_state(S))];
        service[pickup,release,send](s,pn) )

    endproc (* service *)
    endproc (* onecall *)
  endproc (* onecallperphoneconstraint *)

process busyconstraint[send,state] : noexit:=
  busyconstraint1[send,state](make4(1,1,1,1))
  |||
  busyconstraint1[send,state](make4(2,2,2,2))
  |||
  busyconstraint1[send,state](make4(3,3,3,3))
  |||
  busyconstraint1[send,state](make4(4,4,4,4))
where

  process busyconstraint1[send,state](pn:phone_number) : noexit :=
    service[send,state](begin,mhook(off),pn)

  where

    process service[send,state](s:stmstate,S:message,pn:phone_number): noexit :=

      [s eq begin] ->
        ( send !phoneset(pn) !simplug(pn) !mhook(off);
          service[send,state](s1,S,pn)
          []
          state!service!simplug(pn)!sfree; service[send,state](begin,S,pn)
          []
          send !phoneset(pn) !simplug(pn) ?S:message [not(is_state(S))];
            service[send,state](begin,S,pn) )
      []
      [s eq s1] ->
        ( send !phoneset(pn) !simplug(pn) !mhook(on);

```

Figure D.17 Combining slices **switch** level: behaviour **concurrentConnect** (Continued ...)

```

        service[send,state](begin,S,pn)
      []
      state !service!simplug(pn) !slbusy; service[send,state](s1,S,pn)
      []
      send !phoneset(pn) !simplug(pn) ?S:message [not(is_state(S))];
        service[send,state](s1,S,pn) )

    endproc (* service *)
  endproc (* busyconstraint1 *)
endproc (* busyconstraint *)

process onecallperline[open,close] : noexit :=
  oneline[open,close](0 of ILNat)
  |||
  oneline[open,close](SuccNat(0 of ILNat) of ILNat)

where

process oneline[open,close](line:ILNat) : noexit :=
  service[open,close](begin,{ } of setofpn,line)
  where

    process service[open,close](s:stmstate,spn:setofpn,line:ILNat) : noexit :=

      [s eq begin] ->
        ( open!allocator!internal_line(line)?v_spn:setofpn;
          service[open,close](s1,v_spn,line) )
      []
      [s eq s1] ->
        ( close!allocator!internal_line(line);
          service[open,close](begin,spn,line) )

    endproc (* service *)
  endproc (* oneline *)
endproc (* onecallperline *)

process busylineconstraint[answerIL,close] : noexit :=
  service[answerIL,close](begin)

where
  process service[answerIL,close](s:stmstate) : noexit :=
    [s eq begin] ->
      (answerIL !allocator !service ?Line:ILNat [not(Line eqNat noline)];
        service[answerIL,close](s1))
    []
    [s eq s1] ->
      ((answerIL !allocator !service ?Line:ILNat [not(Line eqNat noline)];
        service[answerIL,close](s2))
        []
        (close ?c1:component ?c2:component;
          service[answerIL,close](begin)))
    []
    [s eq s2] ->
      ((answerIL !allocator !service !noline;
        service[answerIL,close](s2))
        []
        (close ?c1:component ?c2:component;
          service[answerIL,close](s1)))

```

Figure D.17 Combining slices **switch** level: behaviour **concurrentConnect** (Continued ...)

```

    endproc (* service *)
endproc (* busylineconstraint *)

(* TIMELINE PROCESS *)

process call
[pickup,dial,ringing,talk,listen,release,state,answerIL,
open,close,send,receive,askcon,destroy,create,connected,
disconnect,connecton,connect_receive,
connect_send,end_service,askIL,deallocateIL]
    (pn1,pn2:phone_number): noexit:=

( let
    spn:setofpn = insertpn(pn1,insertpn(pn2,{ })),
    n1:dialnumber = car(pn2),
    n2:dialnumber = car(cdr(pn2)),
    n3:dialnumber = car(cdr(cdr(pn2))),
    n4:dialnumber = car(cdr(cdr(cdr(pn2))))
    in
    (
    (
    (
    Normalconnection0[pickup,send](pn1)
    )
    >>
    (
    Normalconnection1[send,receive,listen,dial](pn1,n1,n2,n3,n4)
    [> (Disconnection1[release,send](pn1) >> stop)
    )
    >>
    Normalconnection2[create,askcon](pn1,pn2)
    >>
    accept N:compo_ID in
    (
    Normalconnection3[state,askIL](N,spn,pn2)
    []
    (Busy
    [state,connecton,receive,listen,release,send,
    end_service,destroy]
    (pn1,pn2,N) >> stop
    )
    )
    >>
    accept N:compo_ID in
    (
    Normalconnection4[answerIL,connecton,receive,listen,ringing,open](pn1,pn2,spn,N)
    []
    (Noline[answerIL,connecton,receive,listen,release,
    send,end_service,destroy](pn1,pn2,N) >> stop)
    )
    )

```

Figure D.17 Combining slices **switch** level: behaviour **concurrentConnect** (Continued . . .)

```

>>
accept N:compo_ID,line:ILNat in
(
  Normalconnection5[pickup,send,receive,ringing,connected](pn1,pn2,N,line)
  []
  (Disconnectionafter4
    [release,send,disconnect,receive,ringing,deallocateIL,
     close,end_service,destroy]
    (pn1,pn2,line,N) >> stop)
)
>>
accept N:compo_ID,line:ILNat in
(
  (
    (SendSound
      [talk,send,connect_send,connect_receive,receive,listen]
      (pn1,pn2,line,sound1,N) >> stop)
    |||
    (SendSound
      [talk,send,connect_send,connect_receive,receive,listen]
      (pn2,pn1,line,sound2,N) >> stop)
  )
  [>
    (
      (DisconnectionConversation
        [release,send,disconnect,receive,listen,end_service,
         deallocateIL,close,destroy]
        (pn1,pn2,line,N) >> stop)
      []
      (DisconnectionConversation
        [release,send,disconnect,receive,listen,end_service,
         deallocateIL,close,destroy]
        (pn2,pn1,line,N) >> stop)
    )
  )
)
)
)
)

where
process Normalconnection0[pickup,send]
  (pn1:phone_number):exit:=
  pickup !external !phoneset(pn1);
  send !phoneset(pn1) !simplug(pn1) !mhook(off);
  exit
endproc (*Normalconnection0*)

process Normalconnection1[send,receive,listen,dial]
  (pn1:phone_number,n1,n2,n3,n4:dialnumber):exit:=
  receive !simplug(pn1) !phoneset(pn1) !mtone(twaitingdial);
  listen !phoneset(pn1) !external !twaitingdial;
  dial !external !phoneset(pn1) !n1;
  send !phoneset(pn1) !simplug(pn1) !mdial(n1);
  dial !external !phoneset(pn1) !n2;
  send !phoneset(pn1) !simplug(pn1) !mdial(n2);

```

Figure D.17 Combining slices **switch** level: behaviour **concurrentConnect** (Continued ...)

```

dial !external !phoneset(pn1) !n3;
send !phoneset(pn1) !simplug(pn1) !mdial(n3);
dial !external !phoneset(pn1) !n4;
send !phoneset(pn1) !simplug(pn1) !mdial(n4);
receive !simplug(pn1) !phoneset(pn1) !mtone(tsearching);
listen !phoneset(pn1) !external !tsearching;
exit
endproc (*Normalconnection1*)

process Disconnection1[release,send]
  (pn1:phone_number):exit:=
  release !external !phoneset(pn1);
  send !phoneset(pn1) !simplug(pn1) !mhook(on);
  exit
endproc (*Disconnection1*)

process Normalconnection2[create,askcon]
  (pn1,pn2:phone_number): exit(compo_ID) :=
  create !simplug(pn1) !service ?N:compo_ID;
  askcon !simplug(pn1) !service !pn2;
  exit(N)
endproc (*Normalconnection2*)

process Normalconnection3[state,askIL]
  (N:compo_ID,spn:setofpn,pn2:phone_number):exit(compo_ID) :=
  state !service !simplug(pn2) !sifree;
  askIL !service !allocator !spn;
  exit(N)
endproc (*Normalconnection3*)

process Busy
  [state,connecton,receive,listen,release,send,
  end_service,destroy]
  (pn1,pn2:phone_number,N:compo_ID):exit:=
  state !service !simplug(pn2) !slbusy; (*same as create*)
  connecton !service !simplug(pn1) !simplug(pn2) !lbusy;
  receive !simplug(pn1) !phoneset(pn1) !mtone(tbusy);
  listen !phoneset(pn1) !external !tbusy;
  release !external !phoneset(pn1);
  send !phoneset(pn1) !simplug(pn1) !mhook(on);
  (* end_service !simplug(pn1) !service; *)
  (* destroy !simplug(pn1) !service !N; *)
  exit
endproc (*Busy*)

process Normalconnection4[answerIL,connecton,receive,listen,ringing,open]
  (pn1,pn2:phone_number,spn:setofpn,N:compo_ID):exit(compo_ID,ILNat) :=
  open !allocator ?c:component !spn [isrealline(c)];
  answerIL !allocator !service !getILNat(c);
  connecton !service !simplug(pn1) !simplug(pn2) !getILNat(c);
  receive !simplug(pn1) !phoneset(pn1) !mtone(tringing);
  receive !simplug(pn2) !phoneset(pn2) !mringing(on);

```

Figure D.17 Combining slices **switch** level: behaviour **concurrentConnect** (Continued ...)

```

listen !phoneset(pn1) !external !tringing;
ringing !phoneset(pn2) !external !on;
exit(N,getILNat(c))
endproc (*Normalconnection4*)

process Noline[answerIL,connecton,receive,listen,release,send,end_service,destroy]
    (pn1,pn2:phone_number,N:compo_ID):exit:=
    answerIL !allocator !service !noline;
    connecton !service !simplug(pn1) !simplug(pn2) !noline;
    receive !simplug(pn1) !phoneset(pn1) !mtone(tbusy);
    listen !phoneset(pn1) !external !tbusy;
    release !external !phoneset(pn1);
    send !phoneset(pn1) !simplug(pn1) !mhook(on);
    end_service !simplug(pn1) !service;
    destroy !simplug(pn1) !service !N;
    exit
endproc (*Noline*)

process Normalconnection5[pickup,send,receive,ringing,connected]
    (pn1,pn2:phone_number,N:compo_ID,line:ILNat):exit(compo_ID,ILNat):=
    pickup !external !phoneset(pn2);
    send !phoneset(pn2) !simplug(pn2) !mhook(off);
    receive !simplug(pn2) !phoneset(pn2) !mringing(off);
    connected !simplug(pn1) !simplug(pn2) !service;
    ringing !phoneset(pn1) !external !off;
    exit(N,line)
endproc (*Normalconnection5*)

process Disconnectionafter4[release,send,disconnect,receive,ringing,
    desallocateIL,close,end_service,destroy]
    (pn1,pn2:phone_number,line:ILNat,N:compo_ID):exit:=
    release !external !phoneset(pn1);
    send !phoneset(pn1) !simplug(pn1) !mhook(on);
    disconnect !service !simplug(pn1) !simplug(pn2);
    receive !simplug(pn2) !phoneset(pn2) !mringing(off);
    ringing !phoneset(pn2) !external !off;
    desallocateIL !service !allocator !line;
    close !allocator !internal_line(line);
    end_service !simplug(pn1) !service;
    destroy !simplug(pn1) !service !N;
    exit
endproc (*Disconnectionafter4*)

process SendSound[talk,send,connect_send,connect_receive,receive,listen]
    (pn1,pn2:phone_number,line:ILNat,ssound:sound,N:compo_ID):
        exit:=
    talk !external !phoneset(pn1) !ssound;
    send !phoneset(pn1) !simplug(pn1) !msound(ssound);
    connect_send !simplug(pn1) !internal_line(line) !ssound;
    connect_receive !internal_line(line) !simplug(pn2) !ssound;
    receive !simplug(pn2) !phoneset(pn2) !msound(ssound);
    listen !phoneset(pn2) !external !ssound;
    exit

```

Figure D.17 Combining slices **switch** level: behaviour **concurrentConnect** (Continued ...)

```

endproc (*SendSound*)

process DisconnectionConversation
    [release,send,disconnect,receive,listen,end_service,
     desallocateIL,close,destroy]
    (pn1,pn2:phone_number,line:ILNat,N:compo_ID):exit:=
    release !external !phoneset(pn1);
    send !phoneset(pn1) !simplug(pn1) !mhook(on);
    disconnect !service !simplug(pn1) !simplug(pn2);
    receive !simplug(pn2) !phoneset(pn2) !mtone(tbusy);
    listen !phoneset(pn2) !external !tbusy;
    release !external !phoneset(pn2);
    send !phoneset(pn2) !simplug(pn2) !mhook(on);
    (* end_service !simplug(pn2) !service; *)
    desallocateIL !service !allocator !line;
    close !allocator !internal_line(line);
    (* destroy !simplug(pn2) !service !N; *)
    exit
endproc (*DisconnectionConversation*)

endproc (* call *)

endspec (*simple_pots*)

```

Figure D.17 Combining slices **switch** level: behaviour **concurrentConnect**

D.4 Component Specifications

One method of extracting and testing component specifications is by means of the ϕ_c transform. The LOTOS expression $\phi_{switch}(\text{concurrentConnect})$ is shown in figure D.18. This LOTOS expression of figure D.18 was derived by application of the `phi` transform of Appendix E; no simplifications or data extensions were applied. Complete component specifications can be found in [26]

```
(*****  
(** Allocator component specification derived by phi      (**)  
(**   transform from the slice expression                (**)  
(**   version 1.0                                       (**)  
(***)  
(*****  
(* Every part concerning service objects have been put into remarks*)  
(*****  
  
specification phi_alloc_concurrentConnect  
  [answerIL,open,close,askIL,desallocateIL]  
  : noexit  
  
(*****  
(** Behaviour and process specification                (**)  
(*****  
  
behaviour
```

Figure D.18 Expression `phi_alloc` derived from slice expression `concurrentConnect`. (Continued . . .)

```

(
(
call
  [answerIL,open,close,askIL,deallocateIL]
  (make4(1,1,1,1),make4(2,2,2,2))
|||
call
  [answerIL,open,close,askIL,deallocateIL]
  (make4(1,1,1,1),make4(3,3,3,3))
)
|[open,close]|
onecallperline[open,close])
|[answerIL,close]|
busylineconstraint[answerIL,close]

where

(* CONSTRAINT PROCESS *)

process onecallperline[open,close] : noexit :=
  oneline[open,close](0 of ILNat)
  |||
  oneline[open,close](SuccNat(0 of ILNat) of ILNat)

where
  process oneline[open,close](line:ILNat) : noexit :=
    service[open,close](begin,{ } of setofpn,line)
  where

    process service[open,close]
      (s:stmstate,spn:setofpn,line:ILNat): noexit :=

      [s eq begin] ->
        ( open!allocator!internal_line(line)?v_spn:setofpn;
          service[open,close](s1,v_spn,line) )
      []
      [s eq s1] ->
        ( close!allocator!internal_line(line);
          service[open,close](begin,spn,line) )

    endproc (* service *)
  endproc (* oneline *)
endproc (* onecallperline *)

process busylineconstraint[answerIL,close] : noexit :=
  service[answerIL,close](begin)

```

Figure D.18 Expression `phi_alloc` derived from slice expression `concurrentConnect`. (Continued ...)

```

where
process service[answerIL,close](s:stmstate) : noexit :=
  [s eq begin] ->
    (answerIL !allocator !service ?Line:ILNat
      [not(Line eqNat noline)];
     service[answerIL,close](s1))
  []
  [s eq s1] ->
    ((answerIL !allocator !service ?Line:ILNat
      [not(Line eqNat noline)];
     service[answerIL,close](s2))
     []
    (close ?c1:component ?c2:component;
     service[answerIL,close](begin)))
  []
  [s eq s2] ->
    ((answerIL !allocator !service !noline;
     service[answerIL,close](s2))
     []
    (close ?c1:component ?c2:component;
     service[answerIL,close](s1)))

endproc (* service *)
endproc (* busylineconstraint *)

(* BEHAVIOUR PROCESS *)

process call
  [answerIL,open,close,askIL,deallocateIL]
  (pn1,pn2:phone_number): noexit:=

( let
  spn:setofpn = insertpn(pn1,insertpn(pn2,{ })),
  n1:dialnumber = car(pn2),
  n2:dialnumber = car(cdr(pn2)),
  n3:dialnumber = car(cdr(cdr(pn2))),
  n4:dialnumber = car(cdr(cdr(cdr(pn2))))
  in
  (
  (
    (
      Normalconnection3[askIL](spn,pn2)
    )
    >>
    (
      Normalconnection4[answerIL,open](pn1,pn2,spn)
    )
  )
  [Noline[answerIL](pn1,pn2) >> stop]
  )
  >>

```

Figure D.18 Expression `phi_alloc` derived from slice expression `concurrentConnect`. (Continued ...)

```

accept line:ILNat in
(
  exit(line)
  []
  (Disconnectionafter4
    [desallocateIL,close]
    (pn1,pn2,line) >> stop)
  )
  >>
  accept line:ILNat in
  (
    (
      (exit >> stop)
      |||
      (exit >> stop)
    )
    [>
      (
        (DisconnectionConversation
          [desallocateIL,close]
          (pn1,pn2,line) >> stop)
        []
        (DisconnectionConversation
          [desallocateIL,close]
          (pn2,pn1,line) >> stop)
        )
      )
    )
  )
)
)
)
)

```

where

```

process Normalconnection3[askIL]
  (spn:setofpn,pn2:phone_number):exit:=
  askIL !service !allocator !spn;
  exit
endproc (*Normalconnection3*)

```

```

process Normalconnection4[answerIL,open]
  (pn1,pn2:phone_number,spn:setofpn):exit(ILNat):=
  open !allocator ?c:component !spn [isrealline(c)];
  answerIL !allocator !service !getILNat(c);
  exit(getILNat(c))
endproc (*Normalconnection4*)

```

Figure D.18 Expression `phi_alloc` derived from slice expression `concurrentConnect`. (Continued ...)

```

process Noline[answerIL]
  (pn1,pn2:phone_number):exit:=
  answerIL !allocator !service !noline;
  exit
endproc (*Noline*)

process Disconnectionafter4[desallocateIL,close]
  (pn1,pn2:phone_number,line:ILNat):exit:=
  desallocateIL !service !allocator !line;
  close !allocator !internal_line(line);
  exit
endproc (*Disconnectionafter4*)

process DisconnectionConversation
  [desallocateIL,close]
  (pn1,pn2:phone_number,line:ILNat):exit:=
  desallocateIL !service !allocator !line;
  close !allocator !internal_line(line);
  exit
endproc (*DisconnectionConversation*)

endproc (* call *)
endspec (*simple_pots*)

```

Figure D.18 Expression `phi_alloc` derived from slice expression `concurrentConnect`.

Appendix E: Definition of \mathbf{phi}_c Transform

This appendix provides a pseudocode description of the \mathbf{phi}_c transform.

The expression S can be transformed into $\mathbf{phi}_c(S)$ by calling the function $\mathbf{phi}(s)$ (the component name c is assumed to be available as a global variable).

The function $\mathbf{phi}(s)$ simply calls the function $\mathbf{phi2}(s, v)$ where the parameter v represents the set of variables of s which are undefined. Initially this set will be empty, although as $\mathbf{phi2}$ is called recursively, the set v will grow and shrink as events are removed and as variable definitions are added.

The function $\mathbf{phi2}(s, v)$ is organized as a large **case** statement. Depending on the LOTOS operator used to construct s , the appropriate transformation is applied. This transformation must remove all events which represent interactions in which component c is not involved. As these events are removed, it is possible that variable definitions will be removed as well; thus any expression containing these variables will have to be modified.

function Phi
Input: *s*: LOTOS specification --The specification to be transformed.

Output: *T*: LOTOS specification --The transformed specification.

Phi2(*s*, {*v*})
endfunction Phi

function Phi2
Input: *s*: LOTOS specification --The specification to be transformed.
v: List of typed variables --List of currently undefined variables.

Output: *T*: LOTOS specification --The transformed specification.

```

--_*****
--If s is the expression exit, then no change is required.
if
  S = exit
then
  T ← exit

```

```

--_*****
--If s is the expression stop, then no change is required.
elseif
  S = stop
then
  T ← stop

```

```

--_*****
--If expression s is an action prefix of the form e; s' or
-- e [p]; s' and e represents an interaction not involving
-- component c then remove event e, record any variables
-- which become undefined by the removal of e, and apply
-- transform to expression s'.
elseif
  (S = e v1 . . . vm; s' or
  S = e v1 . . . vm [p]; s') and
  component c is not involved in the interaction e v1 . . . vm
  --note that component c may be involved in the interaction whenever
  --one of the vi may evaluate to c by some substitution of the
  --variables.

```

```

then
  --determine the variable definitions being removed.
  forall
    vi of the form ?X:sort
  do
    add X:sort to set V
  end
  --Apply Phi recursively.
  T ← Phi2 (S', V)

--*****
--If expression S is an action prefix of the form e; S' and e
--represents an interaction involving component c then retain c
--as part of the transformed expression.
elseif
  --Determine whether component c is involved.
  --May require operator response.
  S = e v1 . . . vm; S' and
  there exists j such that vj = !E or vj = ?E and
  E may evaluate to c and
  vj represents a component involved in the interaction
then
  vj ← !c;
  substitute c for all instances of expression E in S;
  --Any expressions involving variables of V must be
  --replaced by new variables of the appropriate type;
  forall
    X:sort in V such that there exists vi = !v (X)
    where v (X) of sort sort2 is an expression involving X
  do
    generate new variable identifier Y;
    vi ← ?Y:sort2;
    substitute Y for all occurrences of v (X) in S
  end
  --Apply the Phi transform recursively.
  T ← e v1 . . . vn; Phi2 (S', V)

--*****
--This is the same as the previous case but with a condition.
elseif
  S = e v1 . . . vm [p]; S' and
  there exists j such that vj = !E or vj = ?E and
  E may evaluate to c and
  vj represents a component involved in the interaction
then

```

```

vj ← !c;
substitute c for all instances of expression E in S;
--Any expressions involving variables of v must be
--replaced by new variables of the appropriate type;
forall
  X:sort in v such that there exists vi = !v(X)
  where v(X) of sort sort2 is an expression involving X
do
  generate new variable identifier Y;
  vi ← ?Y:sort2;
  substitute Y for all occurrences of v(X) in S
end
if
  condition p does not contain a variable from set v
then
  T ← e v1...vn[p]; Phi2(S', v)
else
  T ← e v1...vn; Phi2(S', v)

--*****
--The i event is part of every component.
elseif
  S = i; S'
then
  T ← i; Phi2(S', v)

--*****
--If s is of the form [p] -> S' then determine whether p can
--be evaluated. If yes retain it; if no then drop it.
elseif
  S = [p] -> S'
then
  if
    condition p contains a variable from set v
  then
    T ← Phi2(S', v)
  else
    T ← [p] -> Phi2(S', v)

--*****
--Process invocations and definitions.
--Each process invocation results in a potential new process definition.
--Data parameters of the invocation which are expressions containing

```

- free variables must be removed from the parameter list.
- A new process definition is created by applying Φ to the body of
- the process definition, taking into account component names and free variables.
- It is possible for the Φ transform to go into an infinite loop here if process
- P is called recursively, a data parameter x of P is used to compute the
- components involved in an interaction, and different values of x evaluate
- to component c ; in most cases it should be possible to avoid such
- a situation.

elseif

$S = P [\dots] (x_1, \dots, x_n)$ and

definition of process P is:

```
process P [...] (y1:sort1, ..., yn:sortn):exitcondition :=
  S'
```

endproc

then

generate unique process identifier P_c ;

construct lists X_1 and X_2

(and corresponding lists Y_1 and Y_2) such that

list (x_1, \dots, x_n) is an interleaving of X_1 and X_2 and

list $(y_1:sort_1, \dots, y_n:sort_n)$ is an interleaving of Y_1 and Y_2 and

none of the variable identifiers of X_1 are in V and

all of the variable identifiers of X_2 are in V ;

if

the following process definition does not exist

(with a possible renaming of identifiers):

```
process P_c [...] Y1 : exitcondition :=
  Phi2 (S', Y2)
```

endproc

then

define a process P_c such that:

```
process P_c [...] Y1 : exitcondition :=
  Phi2 (S', Y2)
```

endproc

endif

$T \leftarrow P_c [\dots] X_1$;

__*****

– In all other cases make sure that Φ distributes over the

– LOTOS operator, e.g., $\Phi (S1 \parallel S2) = \Phi (S1) \parallel \Phi (S2)$

else

forall

subexpressions S' from which S is constructed

do

compute $\Phi_2 (S', V)$

end

combine the $\Phi_2 (S', V)$ expressions using the same operators

used to combine the s' expressions to build s ; return the
resulting expression as \mathbb{T}
endfunction Phi2

Bibliography

- [1] Wolfram Bartussek and David Parnas. Using assertions about traces to write abstract specifications for software modules. Proc. 2nd Conference European Cooperation Informatics, New York, December 1978.
- [2] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. North-Holland, Amsterdam, 1989.
- [3] T. Bolognesi and M. Caneve. Equivalence verification: Theory, algorithms and a tool. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 303–326. North-Holland, Amsterdam, 1989.
- [4] G. Booch. *Object Oriented Design with Applications*. Benjamin/Cummings, Redwood City, CA, 1991.
- [5] R. Boumezbeur. Design, specification and validation of telephony systems in LOTOS. Technical Report TR-91-30, Dept. of Computer Science - University of Ottawa, September 1991.
- [6] Ed Brinksma. A theory for the derivation of tests. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*. North-Holland, Amsterdam, 1989.
- [7] Ed Brinksma, Giuseppe Scollo, and Chris Steenbergen. LOTOS specifications, their implementations and their tests. In B. Sarikaya and G.V. Bochmann, editors, *Protocol Specification, Testing, and Verification, VI*, pages 349–360. North-Holland, New York, 1986.
- [8] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31:560–599, 1984.

- [9] R.J.A. Buhr. Practical visual techniques for the joint refinement of structure and temporal behaviour in reactive system design. In *Compeuro 90*, Tel Aviv, 1990.
- [10] R.J.A. Buhr. *Practical Visual Techniques in System Design: With Applications to Ada*. Prentice Hall, Englewood Cliffs, N.J., 1990.
- [11] R.J.A. Buhr. Pictures that play for designing concurrent real time systems. Tech. Report SCE-91-08, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, 1991.
- [12] R.J.A. Buhr, G.M. Karam, C.M. Woodside, R. Casselman, G. Frank, H. Scott, and D. Bailey. Timebench: A CAD tool for real-time system design. In *Proceedings of the 2nd International Symposium on Environments and Tools for Ada (SETA2)*, Washington D.C., January 1992.
- [13] J.R. Cameron. An overview of JSD. *IEEE Transactions on Software Engineering*, se-12(2):222–240, February 1986.
- [14] V. Carchiolo, A. Faro, O. Mirabella, G. Pappalardo, and G. Scollo. A LOTOS specification of the PROWAY highway service. *IEEE Transactions on Computers*, C-35(11):949–968, November 1986.
- [15] CCITT. SDL. CCITT Recommendation Z100, 1988.
- [16] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
- [17] A.A.R. Cockburn, W. Citrin, R.F. Hauser, and J. von Kanel. An environment for interactive design of communications architectures. In L. Logrippo, R.L. Probert, and H. Ural, editors, *Tenth International IFIP WG6.1 Symposium on Protocol Specification, Testing and Verification*, pages 107–120, Ottawa, 1990.
- [18] R. DeNicola and M.C. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

- [19]H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification - 1*. Springer-Verlag, Berlin, 1985.
- [20]G. Estrin, R.S. Fenchel, R.R. Razouk, and M.K. Vernon. SARA (System ARchitects Apprentice): Modeling analysis and simulation support for the design of concurrent systems. *IEEE Transactions on Software Engineering*, se-12(2):293–311, February 1986.
- [21]M. Evangelist, N. Francez, and S. Katz. Multiparty interactions for inter-process communication and synchronization. *IEEE Transactions on Software Engineering*, 15(11):1417–1426, November 1989.
- [22]M. Evangelist, V.Y. Shen, I.R. Forman, and M. Graf. Using Raddle to design distributed systems. In *10th International Conference on Software Engineering*, pages 102–107. IEEE, April, 1988.
- [23]M. Faci, L. Logrippo, and B. Stepien. Formal specifications of telephone systems in LOTOS. Technical Report TR-89-07, Computer Science Dept., University of Ottawa, Ottawa, February 1989.
- [24]M.S. Gerhardt and R.R. Dye. A set of steps for doing object-oriented design. 1990.
- [25]Jay L. Gischer. The equational theory of pomsets. *Theoretical Computer Science*, 61:199–224, 1988.
- [26]P. Goldstein, M. Vigder, and R.J.A. Buhr. Case study in LOTOS. Technical Report SCE-91-42, Dept. of Systems and Computer Engineering, Carleton University, 1991.
- [27]R. Gotzhein. The formal definition of the architectural concept ‘interaction point’. In S. Vuong, editor, *Formal Description Techniques (FORTE 89)*. North-Holland, 1989.
- [28]M. Graf. The design of a distributed system using a visual language. Technical Report STP-319-87, MCC, October 1987.

- [29]M. Graf. Building a visual design environment. MCC Technical Report STP-318-87, October, 1987.
- [30]R. Guillemot, M. Haj-Hussein, and L. Logrippo. Executing large LOTOS specifications. University of Ottawa, Dept. of Elec. Eng. report TR-88-03, Ottawa, 1988.
- [31]B.T. Hailpern. *Verifying Concurrent Systems using Temporal Logic*. Lecture Notes in Computer Science 129. Springer-Verlag, New York, NY., 1982.
- [32]M. Haj-Hussein and L. Logrippo. Specifying distributed algorithms in LOTOS. Technical Report TR-91-04, Computer Science Dept., University of Ottawa, Ottawa, 1991.
- [33]Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–20, September 1990.
- [34]D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.
- [35]D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.
- [36]D. Harel and A. Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, NATO ASI Series F: Computer and System Sciences, Vol. 13, pages 3–26. Springer-Verlag, Berlin, 1985.
- [37]D.J. Hatley and I. Pirbhai. *Strategies for Real-Time System Specification*. Dorset House, New York, 1987.
- [38]Richard Helm, Ian Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioural compositions in object oriented systems. In Norman Meyrowitz, editor, *OOPSLA/ECOOP '90*, pages 169–180, 1990.

- [39]D. Helmbold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.
- [40]M. Hennessy. *Algebraic Theory of Processes*. Foundations of Computing. MIT Press, Cambridge, 1988.
- [41]C.A.R. Hoare and J.C. Sheperdson, editors. *Mathematical Logic and Programming Languages*. Prentice-Hall, Toronto, 1985.
- [42]i-Logix, Inc. *STATEMATE Analyzer*. Burlington, Mass., 1988.
- [43]ISO. *Estelle: A Formal Description Technique based on an Extended State Transition Model, DIS9074*. International Organization for Standardization, 1987.
- [44]ISO. *Data Processing - Open Systems Interconnection - Basic Reference Model, IS7498*. International Organization for Standardization, 1988.
- [45]ISO. *G-LOTOS: a graphical syntax for LOTOS, ISO/IEC JTC/SC21 N3253*. ISO, 1989.
- [46]ISO. *Information processing systems - Open systems interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, IS8807*. International Organization for Standardization, 1989.
- [47]I. Jacobson. Language support for changeable large real-time systems. In N. Meyrowitz, editor, *OOPSLA 86*, pages 377–384, 1986.
- [48]I. Jacobson. Object oriented development in an industrial environment. In *OOPSLA 87*, pages 183–191, 1987.
- [49]C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, second edition, 1990.
- [50]P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 228–240, August 1983.

- [51]G. Karam. A tool-set for temporal analysis in a design environment for concurrent systems. Ph.D. Thesis, Carleton University, Ottawa, December 1986.
- [52]G.M. Karam. Mlog: A language for prototyping concurrent systems. Dept. of Systems and Computer Engineering Tech. Report SCE-88-3, Ottawa, November 1988.
- [53]G.M. Karam and R.J.A. Buhr. Starvation and critical race analyzers for ada. *IEEE Transactions on Software Engineering*, 16(8):829–843, August 1990.
- [54]G.M. Karam, C.M. Stanczyk, and G.W. Bond. Critical races in ada programs. *IEEE Transactions on Software Engineering*, 15(11):1471–1480, November 1989.
- [55]P. King and G. Smith. Formalisation of behavioural and structural concepts for communication systems. In L. Logrippo, R.L. Probert, and H. Ural, editors, *Protocol Specification, Testing, and Verification X*, pages 1–18. North-Holland, Amsterdam, 1990.
- [56]C.J. Koomen. Algebraic specification and verification of communication protocols. *Science of Computer Programming* 5, pages 1–36, 1985.
- [57]S. Lamouret, M. Vigder, and R.J.A. Buhr. Semi-automated translation of state machines into LOTOS specification and processes. Technical Report SCE-91-34, Dept. of Systems and Computer Engineering, Carleton University, 1991.
- [58]J.Z. Lavi and E. Kessler. *An Embedded Computer Systems Analysis Method*. November 1986.
- [59]J.Z. Lavi and M. Winokur. ECSAM - a method for the analysis of complex embedded computer systems and their software. In *Structured Techniques Association Conference STA5*, Chicago, 1989.
- [60]G.J. Leduc. The intertwining of data types and processes in LOTOS. In H. Rudin and C.H. West, editors, *Protocol Specification, Testing, and Verification, VII*, pages 123–136. North-Holland, Amsterdam, 1987.

- [61]L. Logrippo, M. Faci, and M. Haj-Hussein. An introduction to LOTOS: Learning by examples. Technical Report Tr-91-21, Dept. of Computer Science - University of Ottawa, May 1991.
- [62]L. Logrippo, A. Obaid, J. P. Briand, and M.C. Fehri. An interpreter for LOTOS, a specification language for distributed systems. *Software - Practice and Experience*, 18(4)(4):365–385, April 1988.
- [63]D.C. Luckham, D.P. Helmbold, S. Meldal, D.L. Bryan, and M.A. Haberler. Task Sequencing Language for specifying distributed Ada systems. Technical Report CSL-TR-87-334, Stanford, CA., July 1987.
- [64]John McLean. A formal method for the abstract specification. *Journal of the Association for Computing Machinery*, 31(3):600–627, July 1984.
- [65]R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, 28. Springer-Verlag, Berlin, 1980.
- [66]A. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, se-7(4), July 1981.
- [67]M. Muhlhauser, A. Schill, H. Frank, and L. Heuser. A software engineering environment for distributed applications. In Lorenzo Mezzalana and Stephen Winter, editors, *Design Tools for the 90's. 15th EUROMICRO Symposium on Microprocessing and Microprogramming (EUROMICRO 89)*. North-Holland, Amsterdam, 1989.
- [68]Max Muhlhauser. Private communication.
- [69]E. Najm and J.-B. Stefani. Dynamic configuration in LOTOS. In K. Parker and G. Rose, editors, *FORTE'91: Fourth International Conference on Formal Description Techniques*, pages 205–222, Sydney, 1991.
- [70]E. Najm and J.-B. Stefani. Object-based concurrency: A process calculus analysis. In S. Abramsky and T.S.E. Maibaum, editors, *TAPSOFT'91 Proceedings of the International Joint Conference on Theory and Practice of Software De-*

- velopment*, Lecture Notes in Computer Science 493, pages 359–380, Berlin, 1991. Springer-Verlag.
- [71]R. De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24:211–237, 1987.
- [72]S.S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.
- [73]J.L. Peterson. Petri nets. *Computing Surveys*, 9(3):223–252, September 1977.
- [74]A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In J.W. de Bakker, W-P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency, Lecture Notes in Computer Science 224*, pages 510–584. Springer-Verlag, New York, 1986.
- [75]A. Pnueli and E. Harel. Applications of temporal logic to the specification of real-time systems. In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science, pages 84–98. Springer-Verlag, New York, 1988.
- [76]Vaughan Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.
- [77]B. Selic, G. Gullekson, J. McGee, and I. Engelberg. ROOM: An object-oriented methodology for developing real-time systems. Technical report, Bell-Northern Research, Ottawa, 1992 (Submitted to CASE '92, Fifth International Workshop on Computer-Aided Software Engineering, Montreal, Canada, July 1992).
- [78]Bran Selic and Jim McGee. Object-oriented design concepts for real-time distributed systems. In *OOPSLA 91*, 1991.
- [79]J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1989.
- [80]K. Turner. An architectural semantics for LOTOS. In H. Rudin and C.H. West, editors, *Protocol Specification Testing and Verification VII*, pages 15–

28. North-Holland, Amsterdam, 1987.
- [81]K. Turner. Tutorial: The role of architecture in formalization. In K. Parker and G. Rose, editors, *Forte'91: Fourth International Conference on Formal Description Techniques*, Sydney, 1991.
- [82]P.H.J. van Eijk. The design of a simulator tool. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 351–390. North-Holland, Amsterdam, 1989.
- [83]P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS*. North-Holland, Amsterdam, 1989.
- [84]C.A. Vissers, G. Scollo, and M. van Sinderen. Architectural and specification style in formal descriptions of distributed systems. In *Proc. IFIP WG6.1, Protocol Specification, Testing, and Verification VIII*, pages 189–204. North-Holland, 1988.
- [85]C.A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, 89:179–206, 1991.
- [86]P.T. Ward and S.J. Mellor. *Structured Development for Real-Time Systems: Volume 1-3*. Yourdon Press, New York, 1985.
- [87]Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, Englewood Cliffs, N.J., 1990.
- [88]P. Zave. The operational versus the conventional approach to software development. *Communications of the ACM*, 27(2):104–118, February 1984.
- [89]P. Zave. A distributed alternative to finite-state-machine specifications. *ACM Transactions on Programming Languages and Systems*, 7(1):10–36, January 1985.