Formalising The Description Of Process Based Simulation Models

Robert John Pooley

PhD
University of Edinburgh
1995

Abstract

Discrete event simulation has grown up as a practical technique for estimating the quantitative behaviour of systems, where direct measurement is undesirable or impractical. It is also used to understand the detailed behaviour of such systems. Its theory is largely that of experimental science. Theories of simulation largely centre on statistical approaches to validating the measures generated by such models, rather than on the verification of their detailed behaviour. This dissertation presents an approach to understanding the correctness of the behaviour of discrete event simulation models, using Milner's Calculus of Communicating Systems (CCS).

It is shown that a common framework based on the process view of models can be constructed for hierarchical modelling, where both performance and functional properties are of interest. As a formal basis for this framework, a hierarchical graphical modelling language (Extended Activity Diagrams) is developed. A semantics is developed for this language, in terms of CCS. This language is shown to map onto the major constructs of the DEMOS discrete event simulation language, extended to allow hierarchical modelling and to resolve certain ambiguities. The result is a new version of DEMOS known as *modified* DEMOS. A graphically driven tool based on such a framework is presented. It allows models to use a combination of simulation and functional techniques to answer both performance questions (what is the throughput under a certain load) and functional questions (will the system deadlock under certain assumptions). In particular this tool can support process oriented simulations of models, using *modified* DEMOS, and functional analysis, based on both the basic version and the timed extension of Milner's Calculus of Communicating Systems and using the Concurrency Workbench. A number of examples of interesting applications of this approach to typical models are presented.

Table of contents

Table of figures

Chapter 1

Introduction

This dissertation contains the resolution of several questions that have been in my mind for over a decade. I had hoped that the major benefit to me in completing this document would be to lay to rest some of them. However, it is in the nature of research that for every issue dealt with, several more spring, Hydra like, to replace them. Perhaps the greatest benefit is really to have built a framework within which these questions can be more clearly addressed and answers assessed.

1.1 The problems addressed

In designing complex systems, simulation is often used to establish both quantitative (performance) and qualitative (behavioural) properties. Its use is, however, expensive and often yields only approximate results. For qualitative properties, Petri nets, process algebras and formal specification techniques are increasingly used. For quantitative properties analytical or numerical modelling, using queues or stochastic extensions to Petri nets, are often preferred. However, simulation remains the only way to handle large models with complex interactions, because of the restricted classes of models suitable for exact solutions and the state space explosion when generating underlying Markov chains for numerical analysis.

Discrete event simulation tools are traditionally categorised as being based on one of a small number of views of a model. A number of modelling tools are based on or can support the process view of simulation as defined by Franta [27]. Several of these, as well as others based on other views, have diagram conventions for users to define their models and some support model construction via graphical interfaces based on such diagrams. Unfortunately, whereas Petri nets generated from graphical tools can be analysed for both functional and performance behaviour, the use of diagrams for simulation is usually specific to one simulation tool and offers no help in

understanding the behaviour of models without actually simulating them. Since discrete event simulation is in effect a (pseudo-)random walk through the state space of the model, it is not possible to guarantee to visit all states without pre-analysis by other means.

The work of this dissertation addresses the problem of developing a formal understanding of process based discrete event simulation models. These are required to be expressible in terms of diagrams suitable for direct graphical input on PCs or workstations. At the same time they must be amenable to a priori functional analysis and so have a well developed semantics. The vehicle for this is the definition of mappings from a graphical language of models (known as Extended Activity Diagrams) both to a discrete event simulation language (an extended form of Birtwistle's DEMOS) [13] and to Milner's Calculus of Communicating Systems (CCS) [58].

A major problem with diagrams for this purpose is that large or complex models are difficult to express and to understand. Fortunately the structure of process based models is inherently hierarchical and so this can be used to provide information grouping and hiding in a natural and consistent manner.

1.2 Overview

This dissertation is structured in the following way. A survey of the main views and their typical diagram conventions is given in Chapter 2. This chapter also contains a survey of previous work in formalising simulation models and in establishing equivalences among them. It concludes with a short description of the Calculus of Communicating Systems (CCS), its temporal extension TCCS [61, 98] and the associated process logic, the modal μ-calculus [95]. Since the main work of the thesis draws on these areas, which are not commonly combined, this initial exposition is quite extensive.

The use of mathematically based formal notation with rigorously defined semantics has many advantages when it is necessary to analyse the properties of systems. Process algebras such as CSP [38] and CCS have evolved for this reason. Unfortunately this way of defining models is often seen as difficult and opaque when presented to practising simulation modellers. Chapter 3 presents a definition of the mechanisms of process based simulation, in particular those in the DEMOS language,

in terms of CCS. This is tested and weaknesses in DEMOS as a vehicle for such definition are identified and remedied, leading to a number of necessary extensions. The way that processes can be decomposed and composed is explored and formalised, leading to a proper understanding of hierarchical, component based modelling which is exploited in Chapters 4 and 5.

Chapter 4 presents Extended Activity Diagrams and their hierarchical extension, Configuration Diagrams, as a basis for describing process based simulation models. The symbols developed match the mechanisms defined in Chapter 3 and so have a definition in terms of both *extended* DEMOS and CCS. A two dimensional grammar for such diagrams is presented, using a slight extension to normal Backus-Naur Form. This allows a rigorous, but abstract, definition of the graphical language, which is independent of any particular physical representation. This plays a key role in simplifying the writing of tools based on graphical input of models.

Chapter 5 presents a tool which supports the ideas in this dissertation. First *extended* DEMOS is described as a set of extensions and modifications to Birtwistle's language. Building on the graphical language defined in Chapter 4, the implementation of a graphical modelling tool is described. This draws mostly on a version developed for IBM PC compatible computers, but also to general solutions to such tool construction. Evidence from a second implementation under X Windows is used to support the possibility of largely automatic tool building based on graphical attribute grammars. The tool is shown to be capable of generating *extended DEMOS* for all models describable using it. Complete CCS models can be generated for many of these models and useful outlines for the others.

Chapter 6 contains some model studies which demonstrate the benefits and problems in combining pre-analysis of functional properties with simulation of dynamic, timed behaviour. It addresses models simplification without changing behaviour, analysis of behavioural properties of models and implications of component model re-use. Not all questions are found to be easily addressed, even with the use of the modal μ-calculus, but some clear benefits are claimed.

Chapter 7 draws together the strands of the earlier chapters and assesses the outcome. Open issues and areas for further research and development are identified.

Chapter 2

Background and previous work

2.1 Introduction

This chapter introduces the background to the work of the thesis. Since it draws on some fields which have had little previous contact, it is somewhat detailed. Those who are familiar with the material of a particular section will perhaps find this unnecessary for themselves, but will hopefully agree to the need in general.

Section 2.2 looks at discrete event simulation and various approaches used to express models to be solved by it. The main purpose of this section is to define clearly the *process based view*, which is the one which will mostly be addressed throughout this dissertation. Section 2.3 considers some of the languages which support the process based view and introduces the DEMOS language [13] as the most comprehensive of these. Section 2.4 is a survey of various ways of drawing discrete event simulation models as diagrams and assesses their suitability as the basis for a formal approach to simulation. Section 2.5 looks at how far *hierarchical* modelling has been addressed within discrete event simulation. Sections 2.4 and 2.5 together motivate the work of Chapter 4 in defining a *complete* and *hierarchical* approach to graphical modelling and Chapter 5 in building a tool to demonstrate this. Section 2.6 considers previous work in formalising discrete event simulation and identifies a lack of rigorous support for behavioural verification of models. Section 2.7 introduces process algebras as a way of expressing behaviour and reasoning about it. In particular the Calculus of Communicating Systems (CCS) [58] is described. Section 2.8 introduces process logics as a means of defining properties of CCS models and posing queries about them. Together sections 2.6, 2.7 and 2.8 motivate and inform the work of Chapters 3 and 6.

2.2 Approaches to discrete event simulation

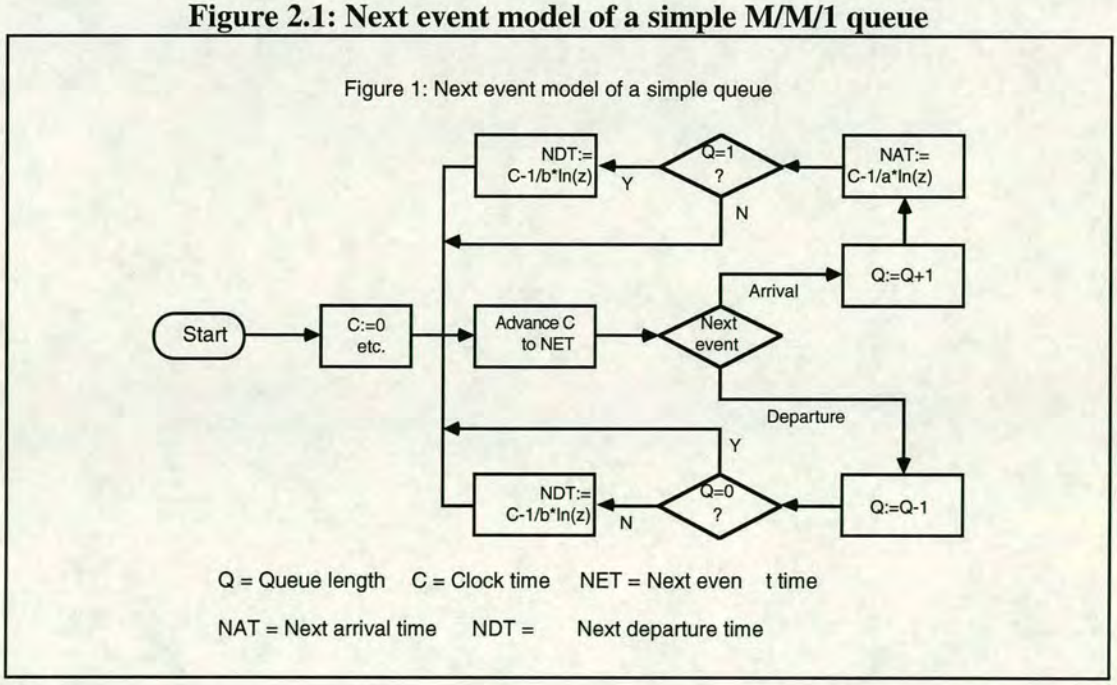
Traditionally, discrete event models are divided into four main world views, *event based*, *activity based*, *transaction based* and *process based*. This dissertation will be mainly concerned with process based models, but a short description of each world view is presented here. For a more extended discussion see [15] and [65].

2.2.1 Event based

Event scheduling is one of the oldest simulation approaches, dating from the 1950s.. It requires models to be viewed as sequences of events through which entities flow, according to various criteria. Such models are similar in many ways to queueing networks. This approach is used in SIMSCRIPT II.5, which is a very popular commercial simulation package. The event scheduling approach is often used to program simple models in general purpose programming languages. It is sometimes argued that event scheduling is more of an implementation device than a conceptually distinct view of modelling, but this ignores strongly held prejudices.

To illustrate the event based approach, consider the following representation of an M/M/1 queue. The model has two types of event, arrivals and departures, both with Poisson rates and uses a next event mechanism. It is clear and easy to code in any standard general purpose language, such as Fortran or Pascal.

Figure 2.1: Next event model of a simple M/M/1 queue



Unfortunately, as described below, such models rapidly become too complex for flowchart representation. Coding them becomes a systematic, but error prone task.

Schruben formalised the event scheduling approach [89], providing a graphical formalism known as *event graphs*. In his original paper he used analytical approaches to explore the behaviour of his models and this was extended in a later paper with Yücesan [109]. This formalisation is considered in more detail below.

Various forms of event graphs have been used to generate discrete event simulation models [66], [86], [39], [54] and [90]. In the last of these, Schruben presents Sigma, a graphical tool for modelling with event graphs. Som and Sargent [93] also present a formalisation of event graphs.

2.2.2 Activity based

Activity based simulation modelling uses a resource centred description of a system, where entities pass through activities. It is often built around graphical notations, such as activity wheel diagrams [97] or activity cycle diagrams [21] [34]. Certain entities are found to pass through cycles of activities, often repeatedly. These cycles are created by the resources of the system.

Activity scanning models are structured along the lines of the physical arrangement of the systems they represent. This has made them popular in applications such as factory simulation. They are similar to Petri nets (see below) in their failure to distinguish active elements from passive, modelling both resource flow and control flow identically.

The activity scanning approach is often inefficient as a means of executing models, since it requires all activities to be checked on each state change to see if there is any effect on entities at that stage. Although this can be improved, it is arguable that activity scanning is inherently inefficient.

2.2.3 Transactions based

The transactions based approach is essentially derived from GPSS [88]. It is more structured than the activity scanning approach and distinguishes between active and passive model components, introducing resources as an explicit modelling concept.

Although GPSS continues to be widely used, it represents a precursor to the process oriented approach. Much of the analysis of behaviour applied below to process oriented models can also be applied to transactions based models.

2.2.4 Process based

The process based view takes as its starting point the idea that the world consists of active and passive components. Although the term was in common use for several years before the appearance of Franta's book [27], this gives the first complete description of the approach, using SIMULA as the implementation language.

Active components (*processes*) are described by their life histories, which often form cycles. They interact with the world through resources, which are passive, in competition or co-operation. This division into two classes is acknowledged to be arbitrary and Franta gives examples where the same object may be seen as active or passive, according to the perspective of the modeller.

The main benefit claimed for the process based approach is that it expresses the model in terms of the structures observable in the real world and so makes modelling more intuitive and interpretation of results easier. It also can have significant implementation benefits, as shown below in the description of SIMULA.

Recently the needs of parallel simulation have led to restrictions on the process view, to remove direct pre-emption of one process by another [22]. This modified process interaction world view actually seems to be an implementation driven one, rather than an improvement in the descriptive powers of the language proposed, but may become accepted if the benefits of parallelisation are seen as desirable.

2.2.5 The relationship among these views

As shown by [15], there is no difference in the set of models which can be expressed in each of these views. Authors have argued with the assertion [27, 13] that the process view is the most natural. Some have argued for attempting to combine their benefits [65]. Here the expressiveness of process based models is assumed initially and then shown to be additionally convenient when using process algebras to formalise our models. On the other hand, the weakness of the interleaving view taken by process based simulation is identified by Evans [26] and more explicit representation of concurrency suggested, using a Petri net based approach.

2.3 Languages for process based simulation

Many languages and packages claim to be process oriented or to be capable of representing process oriented models. Rather like the term *"object oriented"*, process oriented has become a victim of its own success in appealing to ease of understanding. There are, in fact several genuine languages for this purpose. This work will refer mainly to the DEMOS package, which is an extension of SIMULA.

2.3.1 SIMULA

SIMULA [12, 74] is a general purpose programming language, defined as a superset of Algol 60. It was designed to support the efficient implementation of event and process based discrete event simulation. Descriptions of how it may be used in this way are given in [12], [27] and [59] amongst others.

In SIMULA the notion of a process is supported by a combination of inheritance and quasi-parallel sequencing (co-routines or light weight processes) within the class concept. This provides an efficient implementation of conditional waiting, since objects suspended as co-routines can wait in heterogeneous lists and can resume themselves when events in the execution of the model allow them to proceed.

SIMULA supports layers of packages, each refining and extending earlier ones. In this way, a package for list handling, known as SIMSET, is provided on top of the basic language. Using SIMSET, a further layer, known as SIMULATION, is provided. This has a time ordered event list and a class PROCESS, which is the building block for active components in models. PROCESS adds modelling related abstractions to co-routine semantics of classes, in co-operation with the event list.

Although SIMULA does not provide the concept of a general *wait-until* in these packages, Vaucher showed how this could be efficiently implemented within SIMULA by using the Algol *name* mode for procedure parameters [104, 105].

2.3.2 DEMOS

DEMOS [13] is a process oriented discrete event simulation package, written in SIMULA. It does not use the predefined package SIMULATION, but re-implements the event list mechanisms in a similar way.

In addition to those features expected in SIMULA it has automatic statistical collection and reporting and optional output of event traces. In this way, it allows a wide range of models to be solved to establish their dynamic behaviour, both in terms of quantitative performance (response time, queue lengths etc.) and event based behaviour traces.

It offers an efficient version of Vaucher's wait-until mechanism, using an extended version of PROCESS, named class ENTITY and a conditional queue class, CONDQ. Also, a number of more specialised building blocks for the passive elements of a model are provided, all of which report key statistics automatically. These include RES, for resources, BIN, for unbounded buffers, and WAITQ, for master/slave interactions.

DEMOS is investigated extensively in the following chapters.

2.3.3 Alternatives to DEMOS

There are a number of packages offering some of the same features as DEMOS.

GPSS [88] is a restricted form of process based modelling and has influenced the design of DEMOS [15]. In particular it introduced explicit representation of resources as a means of process interaction.

SLAM II [82] is the most widely used alternative. It supports the modelling of processes and their interaction through resources. It lacks the inheritance and co-routine features of SIMULA and DEMOS and so is less extensible, although it offers a wide range of pre-defined options. SLAM II uses *network diagrams* [94], which are described below. It does not support hierarchical modelling.

MODSIM claims to be the most advanced, object oriented simulation language available. It offers somewhat similar features to SLAM, plus some additional flexibility. Its environment includes graphical display features, but it is less flexible than SIMULA in other ways.

2.4 A comparison of existing graphical formalisms

The use of diagrams to describe discrete event models is examined next. The range of approaches in use today is surveyed, with the same simple example given using several different conventions. Criteria by which we can judge the effectiveness of such approaches are suggested and the characteristics of a generally useful standard are developed. In Chapter 4, the core of such a standard is proposed.

2.4.1 General

The use of diagrams to design programs is almost as old as programming itself, predating high level languages as a means of abstraction, for instance in flowcharts. In general this approach has suffered from two practical difficulties:

- the permanence and, hence, difficulty of correction or extension of drawings on paper;

- the explosion of detail in large programs.

Changing diagrams need no longer be a problem, as graphics workstations can be used to create and edit diagrams quickly and cleanly. This opens the way for a wide range of tools to allow *visual* or *graphical programming* [83]. The current popularity of CASE tools shows that this has happened.

The complexity of real models has led to a tendency for diagrams to be used to specify only the high level structure of a program, not the low level implementation. More ambitious approaches have introduced the concept of hierarchical structuring, to allow more detail. This has been easiest where structured programming techniques are already in use or, increasingly, where object oriented programming is being adopted.

The use of diagrams allows a natural expression of parallel activities, actually showing them side by side. This is generally easier to comprehend than the sequential laying out of parallel components in conventional, textual programming languages. Formal notations for the description of concurrent systems, such as Hoare's *Communicating Sequential Processes* [38] and Milner's *Calculus of*

Communicating Systems [107, 58] are examples of attempts to describe parallel systems which use diagrams to aid understanding.

Within discrete event simulation the use of diagrams for tutorial purposes and for program design has long been popular. Until comparatively recently, such diagrams have usually been translated into programs manually. A survey of some of the most widely used of these systems is given below. More recently, several simulation packages have emerged which use graphical input to aid program generation. These packages are typically:

- based on a set of icons for a *single application*, like Simfactory [103];

- based on a low level description, such as *queueing networks* [52, 43] or *Petri-nets* [101, 7, 62,55, 18];

- oriented towards a *particular modelling tool*, like TESS [94], PAWS/GPSM [44] or its successor SES Workbench and PIT [72, 6].

The main aim in suggesting a standard representation is to avoid divergence among such tools. This requires a paradigm which is not predicated on one particular solver or class of solvers, but which can represent models in terms of the systems they are intended to simulate and allow model generation into as many executable forms as possible. The feasibility of such an approach in textual dialogue systems was shown by Mathewson [56]. HIT [8,10] is another example, which now has a graphical interface.

Desirable properties of these techniques are implementation independence, abstraction in terms of the system modelled and completeness of description.

2.4.2 A simple system to model

In order to give some idea of the sorts of diagram which have been used or are in use, it seems best to present a simple model in several of them. Although this is intended to provide both an historical overview and a representative survey, it should not be considered exhaustive.

The model is chosen as typical of the simple, yet realistic, examples given in most introductory texts and, indeed, a version of this model is included in texts for at least three of the tools illustrated. It is important that the example be fairly simple as more complex examples rapidly become too involved to be readable as a diagram. This problem and its solution by *hierarchical decomposition* is described in Chapter 4.

Following example 3.5 in "A System for Discrete Event Modelling on SIMULA" [13], a harbour model is introduced. This is a simpler version of the "African tanker model" originally presented in GPSS terms [88] and later in Slam II terms [81]. The model depicts the life history of a series of ships as they enter a harbour, unload and depart. This is shown as three phases of activity:

docking - which first requires acquisition of a jetty and two tugs and then a delay of known characteristics; the tugs are released at the end of this phase, but the jetty is retained;

unloading - which is a delay of known characteristics, retaining the jetty;

leaving - requires acquisition of a tug, then a delay of known characteristics; once complete, tug and jetty are released and the ship process terminates.

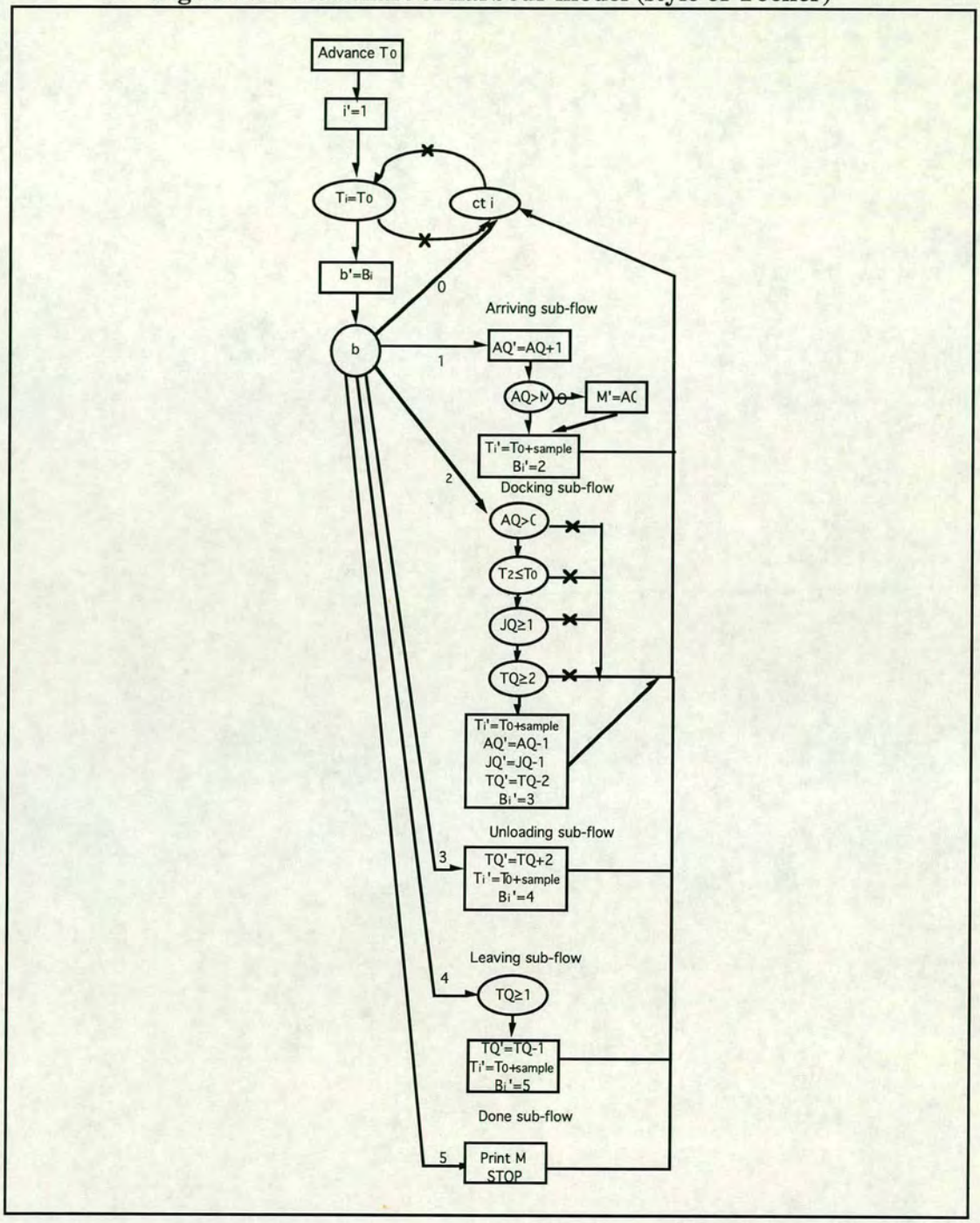
A process oriented model based on this could consist of the instantiation of resources, to represent the tugs and jetties in the harbour, and of new ship processes, at intervals matching their arrival times. The numbers of tugs and jetties and the inter-arrival time distribution of ships would be parameters of this model.

The rest of this section looks at some commonly used diagramming techniques used on this simple system. This then motivates the need for a more general way of describing models in a process based form.

2.4.3 Flowcharts

Originally most programmers, including the author, learned to use flowcharts in their first attempts at programming. For early discrete event simulation modelling, developed in the context of operations research, they were widely used to describe simple models. The example in figure 2.1 is a typical example of such a model presented in that form.

Figure 2.2: Flowchart of harbour model (style of Tocher)



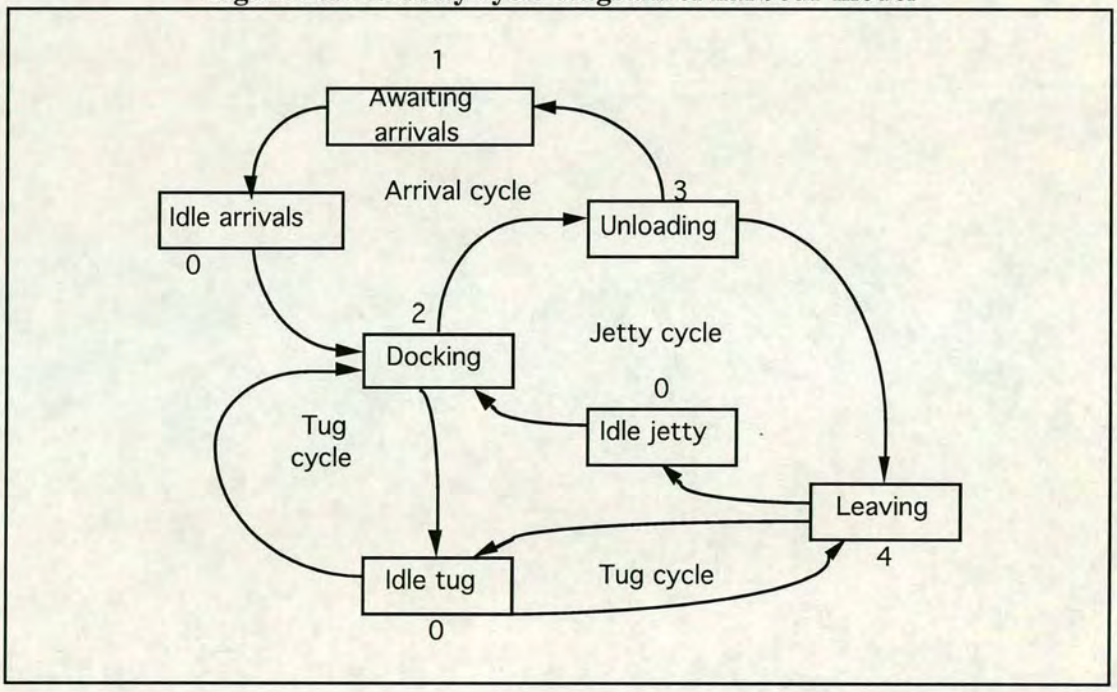
Unfortunately, such models rapidly become too complex to retain their clarity, largely since most general purpose languages offer little support for mechanisms such as event list manipulation. As a result, the need to include the underlying scheduling support obscures the function of the model as a representation of a real

world system. As was pointed out by Tocher early on [97], this scheduling mechanism is often largely the same between models. Figure 2.2 shows the harbour model as a flow chart, using the general style presented by Tocher. Subsequent development of simulation diagramming techniques has generally tried to free system description from implementation.

2.4.4 Activity cycle or wheel diagrams

To improve the ease of specifying a model, as opposed to the corresponding program, Tocher [97] introduced *activity cycle* or *wheel* diagrams. These focused on the cycles of activity associated with components of the system to be modelled. In particular they were used to define all the states which those components could achieve and to show where these interlocked.

Figure 2.3: Activity cycle diagram of harbour model



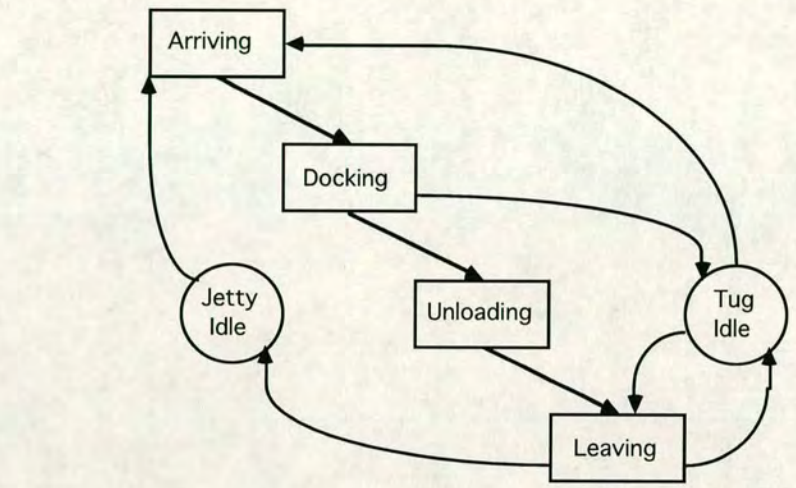
Using this approach, the harbour is shown in figure 2.3. Notice how the cycles represent the sequences of states for what will be termed *resources* in some later versions of this model. The ships themselves need not be shown, as they flow through the system in a rather passive way, although it is possible to add a ship cycle quite easily. In this style of modelling the ships would usually be termed *entities*.

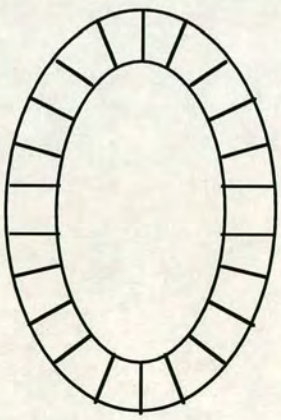
Tocher saw this style of diagram as a stage in defining a flow chart for the program. It lacks much detailed information, although that could be added quite easily.

2.4.5 Hocus activity cycle diagrams

Figure 2.4: Hocus representation

a: Hocus diagram of harbour model





HOCUS queue "track"

ACTIVITY		TIME ACTIVITY ENDS		2 OF 3	
ACTIVITY NAME		NUMBER	NEXT ACTVITY TESTED	IF ACTIVE	3
DOCKING		3		IF PASSIVE	2
SWITCH	SOURCE Q	ENTITY NAME	DEST Q	ALT DEST Q	SWITCH
B+	ARRS	SHIP1	UNLOAD		A-
	T_IDLE	TUG 1	T_IDLE		
	T_IDLE	TUG 2	T_IDLE		
	J_IDLE	JETTY 1	UNLOAD		
CONST	DATA FIELD		COEFFICIENT		%TIME
3	5		0.1		RECRD

HOCUS activity table

b: Hocus basic symbols

The activity style of modelling is still very popular and a number of packages to support it have been produced such as ECSL [21]. An interesting variation on the activity cycle approach was devised by Hills [34,35] and is marketed by PE

Consultants. Known as Hocus (Hand Or Computer Simulation System), its diagram based models can be solved by hand or coded for solution by computer. This is a rare example of a completely diagram based approach to non-computer simulation. It is rather like a board game when approached in this way.

Figure 2.4a shows a Hocus version of the harbour model. It is very similar to the wheel diagrams of Tocher, but now distinguishes two possible states for an entity. The circles are idle states in the model, which are to be programmed as queues. The rectangles are busy states, where the entity is engaged in an activity.

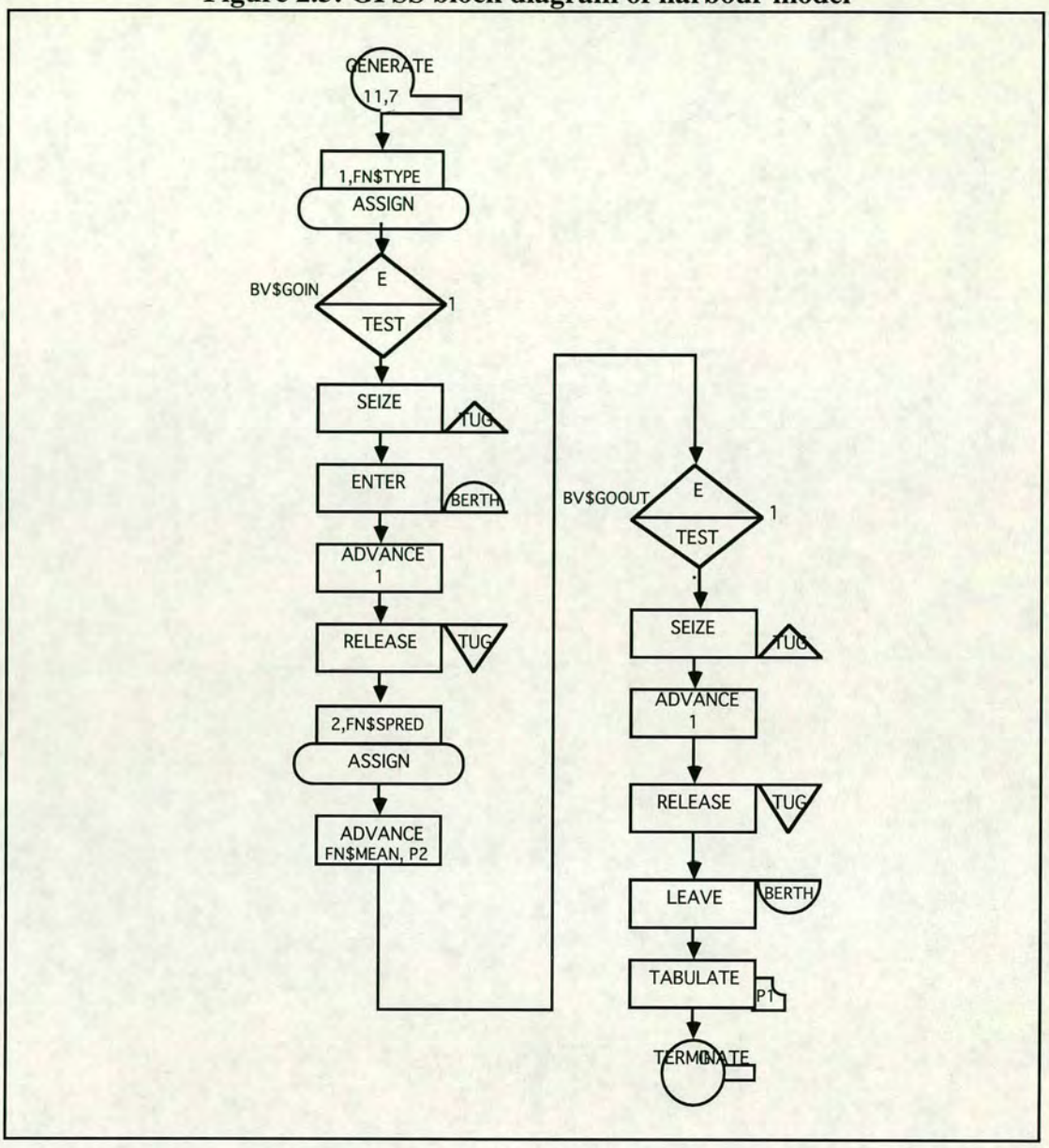
In hand simulations, counters or flags are moved to different states to correspond to the flow of entities. As with the wheel diagrams, some entities remain in the system and can be regarded as resources or servers, while others flow through the system and can be regarded as jobs or customers. To support hand simulations an annotated version of the Hocus symbols is used, where the circles became ovals of cells. Each cell could hold one entity and the continuous "track" of cells minimised movement of counters when an entity left the queue. The activity symbols were tables recording times of events and details of entities engaging in them. These detailed symbols are shown in figure 2.4b.

2.4.6 GPSS transaction block diagrams

GPSS [88] is one of the best known and longest serving simulation packages. It has evolved considerably over the years, but the approach embodied in its *block diagrams* remains its core. The diagram of the harbour model shown in figure 2.5 is a simplified version of figure 6A.1 in Schriber's book. In essence, each block in the flowchart-like diagram corresponds to one GPSS statement.

Notice that this time the story is told from the point of view of the ships. An initialisation segment generates the start conditions and a generator pumps new ships into the model. The main activities are shown as time *Advance* statements, representing the delay involved. These are synchronised with other ships through actions called *Seize* and *Release*, which involve the *Resources* in the model.

Figure 2.5: GPSS block diagram of harbour model



GPSS is the first approach which corresponds in a limited way to the process view of simulation. Although GPSS block diagrams resemble flowcharts, they have a *system based* view of the model, rather than the *program based* view of simple flowcharts in Figures 2.1 and 2.2 above. Thus they are an abstraction towards the problem domain. Their main drawbacks are the degree of detail that is shown, or rather the number of steps required, to achieve simple ends, and the difficulty of showing interactions between different kinds of process in the same model. Although not significant in this simple model, which has only one kind of process, this soon becomes essential. In fact GPSS allows the modeller to specify several processes as

separate diagrams and to associate them through resource names. This is adequate, but lacks the clear expression of interlocking sequences possible with activity cycle diagrams.

2.4.7 Simscript diagrams

Along with GPSS and SIMULA, Simscript [50,51] is one of the earliest widely available languages with specific support for simulation. There seems to be a shortage of generally published recent documentation on this system, in particular the references found for version II.5, which contains the notion of processes, seem not to be generally publicly available [51,84,85]. The example of a model in Simscript terms is based on diagrams used in a paper [25] which may not reflect current usage.

In essence Simscript uses *events*, *sets* (queues) and *routines*. Its original view was based on events rather than processes or activities. The current version is said to contain support for both processes and resources, following the pattern set by GPSS.

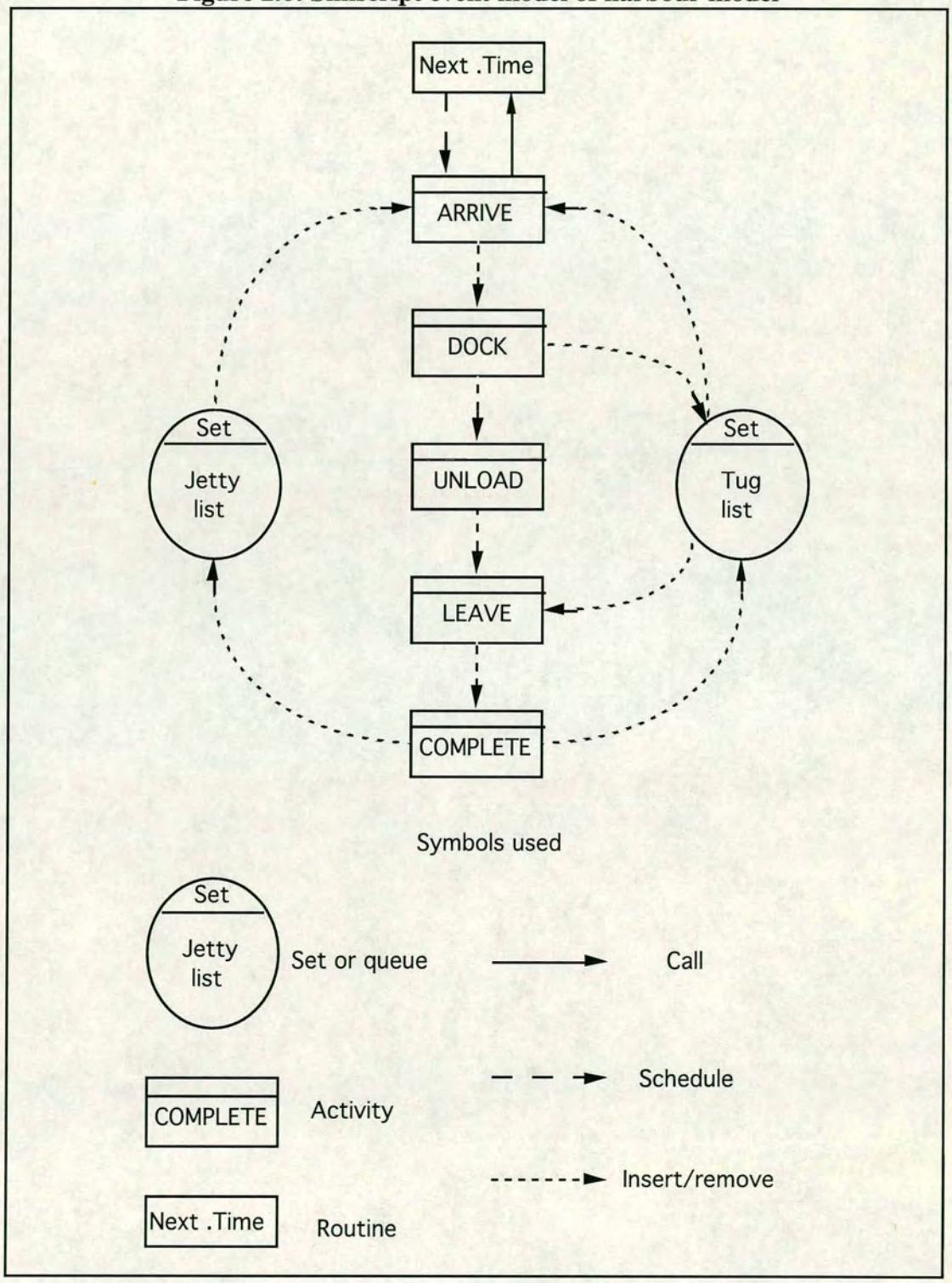
Various types of link are possible between events and other components. These are basically

- *scheduling* of one event by another, shown as a dashed line;
- *insertion* and *removal* of entities in sets, shown as a dotted line;
- *calling* of routines, shown as a full line.

Figure 2.6 shows such a representation of the harbour. In a simple example, where only one type of process is represented, this looks little different to many process based diagramming techniques. Note, however, that the dashed lines between events represent scheduling rather than flow of control. Thus, delays are modelled as explicit scheduling actions. As will be shown in Chapter 3, this is not inconsistent with the process based approach, but weakens the modularity of systems, making component identification more difficult.

One aid to clarity in this form of diagram is the use of links to show flows in and out of sets. These are not just used as resources, however, and in more complex examples confusion amongst different uses of sets is a problem.

Figure 2.6: Simscript event model of harbour model

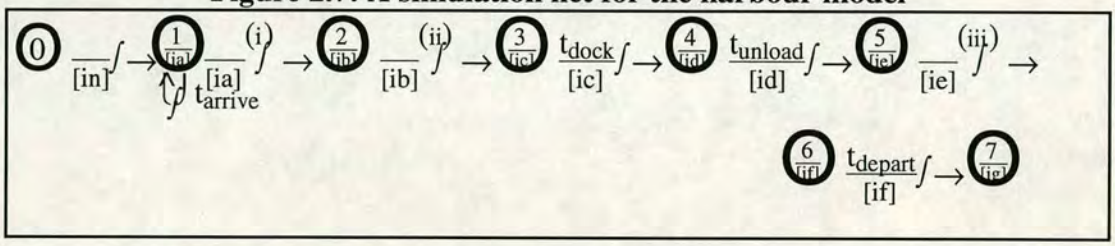


2.4.8 Simulation Nets and Simulation Graphs

Schruben [89] introduced a graphical representation of event based models, which he called *event graphs*. This was extended and formalised in work with Yücesan [109], and called *simulation graphs*. These were used to explain certain formal rules of the behaviour of such models and to prove certain conditions for equivalence and for event reduction within such models. This formal work is contrasted with the use of process algebras in Chapter 6 of this dissertation.

Each node in the directed graph represents a state change or event. The edges in the graph represent the *triggering* (or in the case of a dotted line *cancelling*) of the destination event. Each edge is labelled with an optional time, an optional condition (shown as a roman numeral in parentheses) and an optional *edge attribute list*. The edge attributes are associated with vertex parameters in their destination vertices and represent the passing of parameters on the triggering of an event. This allows, for instance, an instance of an event to be defined as relating to one particular entity in the model. In the model below we use it to identify which ship is in which state at any time.

Figure 2.7: A simulation net for the harbour model



Event Type	Event Description	State Changes
0	Initialisation	Jetties := 2, Tugs := 3, in := 1
1	Ship arrives	ia := ia+1
2	Ship acquires one jetty	Jetties := Jetties - 1
3	Ship acquires two tugs	Tugs := Tugs - 2
4	Ship ends docking	Tugs := Tugs + 2
5	Ship ends unloading	
6	Ship acquires one tug	Tugs := Tugs - 1
7	Ship has left	Tugs := Tugs + 1, Jetties := Jetties + 1

Although some states shown here have no state changes, that is because there is no statistical recording, which would need to be added by showing queues growing and shrinking and times elapsing for sequences of events being recorded. The conditions associated with this graph are as follow:

$C_{1,2}$	[i]	Tugs >= 2
$C_{2,3}$	[ii]	Jetties >= 1
$C_{5,6}$	[iii]	Tugs >= 1

Analysis of this by Schruben and Yücesan's rules is considered in section 2.6.2.

2.4.9 PAWS queueing networks

The Performance Analyst's Workbench System (PAWS) [43] contains a language for performance modelling of information processing systems. It uses a version of queueing network diagrams known as *Information Processing Graphs* (IPGs) and, through its Graphical Programming of Simulation Models (GPSM) interface, allows direct program entry in that form on IBM PC compatible machines. SES Workbench is a development of PAWS and runs on SUN workstations under X Windows. The harbour model is shown as an IPG in figure 2.7. Like Simscript II.5, little generally published material seems to be available on PAWS and this section is derived from the user manuals for the system, in particular the GPSM manual [44]. Some examples of the use of IPGs are given by Smith [92].

Like SLAM II, PAWS has resources as explicit features, which may be Allocated, Released, Created and Destroyed. In addition it has a class of *memory resources*, to model memory in computer systems. The latter allow arbitrary blocks of a resource to be taken from a pool.

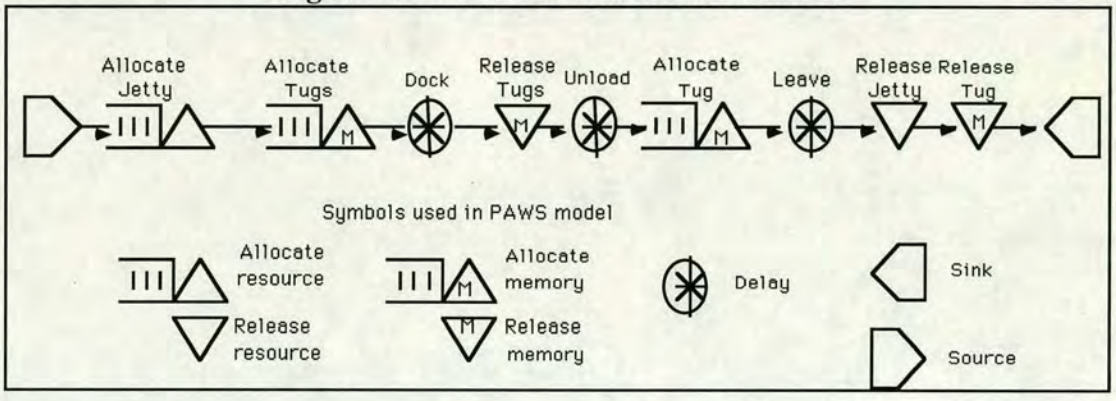
Activities are modelled as either delays (corresponding to infinite server queues in queueing networks) or servers attached to queues.

As in GPSS, the processes in a system are called transactions and these flow through the network to represent the model running. At various nodes transactions may be generated (Source), be destroyed (Sink), change state (Phase Change) or be spawned by parent transactions (Fork) or by sibling transactions (Split). Probabilistic branching is also supported as in classic queueing networks. Forked children may co-alesce into their parent at Join nodes, which violates normal product queueing

network assumptions, but can be easily simulated. Nodes are available for computation, to update state variables.

More interestingly for the purposes of this dissertation, IPGs support the notion of one activity interrupting another and forcing it to shift to another sequence of actions, specified by some parameter. This is clearly a result of PAWS' orientation towards modelling computer systems, which typically have a hardware with interrupts built in. Such a mechanism is also useful for modelling breakdowns, but will be seen to cause considerable problems in DEMOS, when a formalisation in CCS is required.

Figure 2.8: PAWS IPG of harbour model



Also interesting is the support for some notion of hierarchy in IPGs. This is quite natural in a queueing network based system, where the notion of *flow equivalent sub-networks* is a common technique to make large models more easily tractable. In PAWS it is more a question of allowing suitably sized models to be generated, both graphically and for solution. In effect any part of the total IPG network which has a single entry and exit point is a candidate for collapsing into a single node, representing a sub-model to be called when it is reached. This notion of decomposition is more restrictive than that developed in Chapter 4. It allows more complex models to be supported, but is not tied to any explicit structure of the system being modelled.

In general PAWS fails to allow abstraction towards the problem domain, except when suitable sub-models are definable. Instead it provides a way of using a few higher level abstractions, such as resources, to ease the task of building models which are solution method oriented. In the case where analytic methods are to be applied to solve such models this may be necessary, but PAWS is intended as a

simulation package and such a representation tends to obscure the model for non-experts.

2.10 Petri-nets

Petri-nets are among the lowest level representations of a model. It has been suggested, by Hughes for example, that many of the other forms of representation described here could be transformed into equivalent Petri-nets. Some incomplete work on systematically transforming DEMOS style activity diagrams (see 2.4.12 below) into Petri nets was reported in private discussion with Peter Hughes to have been performed by a masters student at the University of Trondheim. Unfortunately it has proved impossible to obtain a written report of this work. Several forms of Petri-net, such as stochastic nets and timed nets, have been devised to allow more complete description of the information needed by model solvers. The harbour model is considered using a simple *place/transition net* .

In general Petri-nets represent models in terms of *tokens* which flow along the edges (called *arcs*) of a directed graph. The nodes are called *places* (shown as circles) and *transitions* (shown as vertical or horizontal bars). Tokens accumulate in places until the satisfaction of some condition associated with a transition on an output arc of that place causes it to fire. When a transition fires it sends the tokens from its input places on along its forward arcs to its output place or places.

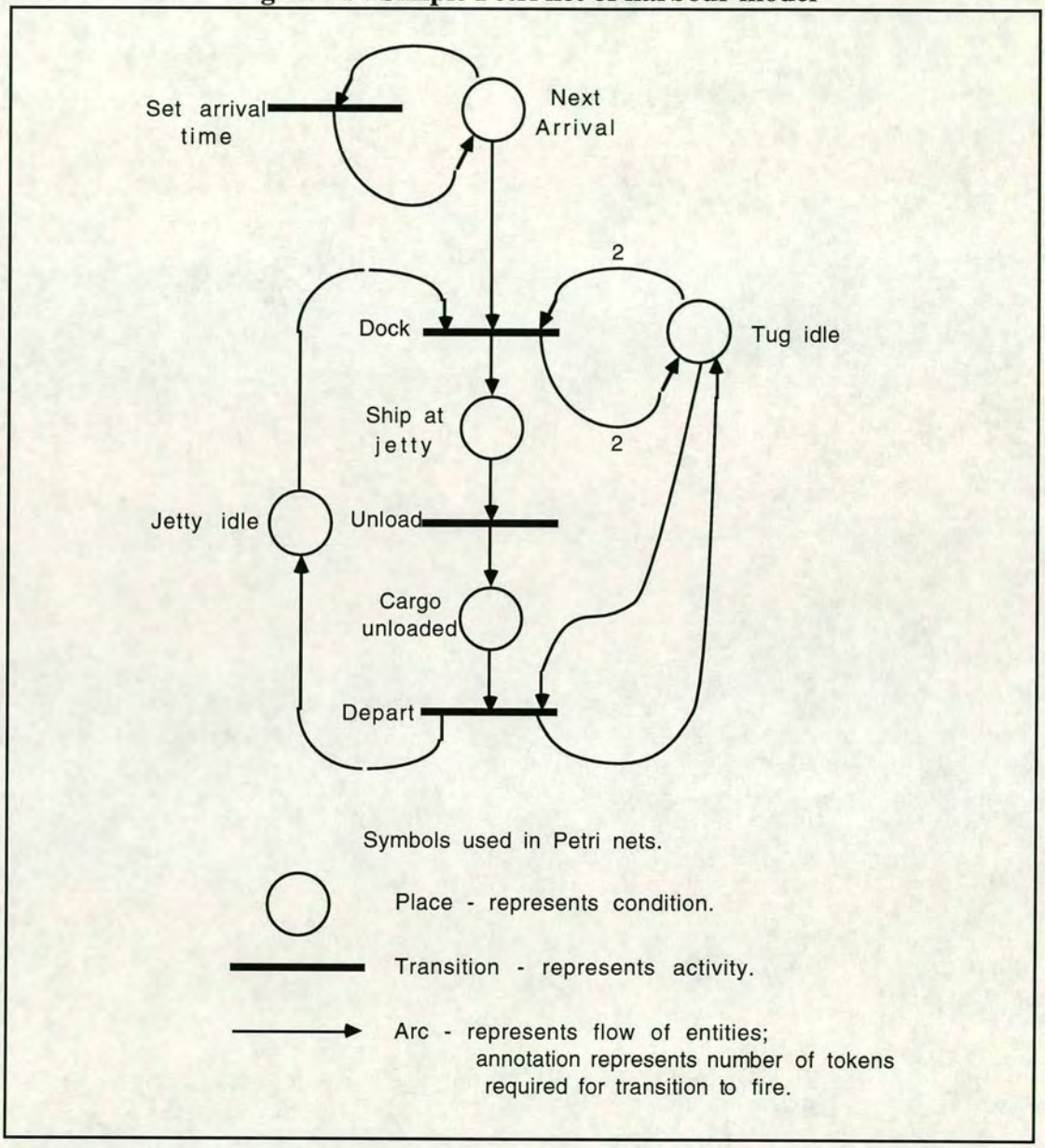
The transitions can be used to represent activities, like the links in SLAM II networks. The nodes represent synchronisations. Timed nets allow durations for transitions to be specified. Stochastic nets allow probabilities to be attached to firings.

Figure 2.9 shows the harbour model as a simple place/transition Petri-net. Note that the scheduling of arrivals must be modelled as a clock loop at the start of the graph and that the model has no implicit timing information or probabilistic behaviour.

Petri-nets have been widely favoured by some modelling theorists, because of their universality and their sound theoretical foundations. They are, however, sometimes quite hard to interpret as system descriptions, as can be seen by the failure to distinguish tokens which represent resources, tokens which represent active

processes and tokens which represent active processes which have acquired resources.

Figure 2.9: Simple Petri net of harbour model



It is, however, straightforward to build interpreters to "execute" or simulate them, but the results may be hard to relate to the structure of the real system being modelled. Clearly they are important in advancing the understanding of modelling, but they are not appropriate, perhaps, as a user interface for non-experts. They can express genuine concurrency, but do not solve the problem of how to program this for all cases when producing an executable discrete event simulation model of the

system which they describe. This problem of expressing concurrency is a major one for all the systems described here. An attempt to unify Petri-nets and process based modelling is made in the *engagement strategy* [26].

Molloy [63] introduced stochastic timings into his models as delays on the firing of enabled transitions and Ajmone Marsan and others [2] introduced more general stochastic modelling mechanisms, most importantly the concept of an *immediate transition*, which took no time but could have branching probabilities associated with output arcs, and inhibiting arcs, which blocked their output places rather than enabling them. The resulting class of generalised stochastic Petri nets (GSPNs) has attracted much interest in the performance analysis community, especially since, where all delays are exponentially distributed, efficient numerical solution techniques are sometimes possible. In such models, the GSPN can be transformed into its equivalent Markov chain.

Deterministic timed delays have also been introduced and efficient numerical methods for solving the resulting *deterministic and stochastic Petri nets* (DSPNs) have been developed by Lindemann [55]. More recently Chiola and others have proposed the use of *coloured Petri nets* to make modelling easier and help with the expressiveness of Petri nets. This work is still in its early days but should remove the problems in distinguishing different uses of tokens flowing through the system.

GreatSPN [18] and Molloy and Riddle's system [62] allow graphical entry of Petri-net models. GreatSPN uses numerical solution methods as well as simulation. It provides several forms of structural and behavioural analysis of models, which will be examined in more detail in section 2.6 below. Lindemann's DSPNExpress [55] is a re-working of GreatSPN for DSPNs. The graphical description of the harbour model remains essentially unchanged, except that timed transitions are used, shown as boxes.

2.4.11 Slam II network diagrams

There have been several published models using SLAM II and its associated network diagrams [82]. These seem to combine some aspects of queueing network diagrams and Petri-nets with GPSS-like resources and activities expressed as delays. Pritsker also used these ideas to show Q-GERT models [81], which include continuous state changes.

The immediately obvious difference in figure 2.10 is that activities are shown on links of the diagram, not as nodes. They are rather similar to timed Petri-nets (see section 2.4.10 above), although in general a SLAM II description is much more system oriented. The model shown follows what seems to be the normal convention in network diagrams by showing flow through the model as horizontal, generally left to right. This may make them seem rather more different from some others shown here than is really the case.

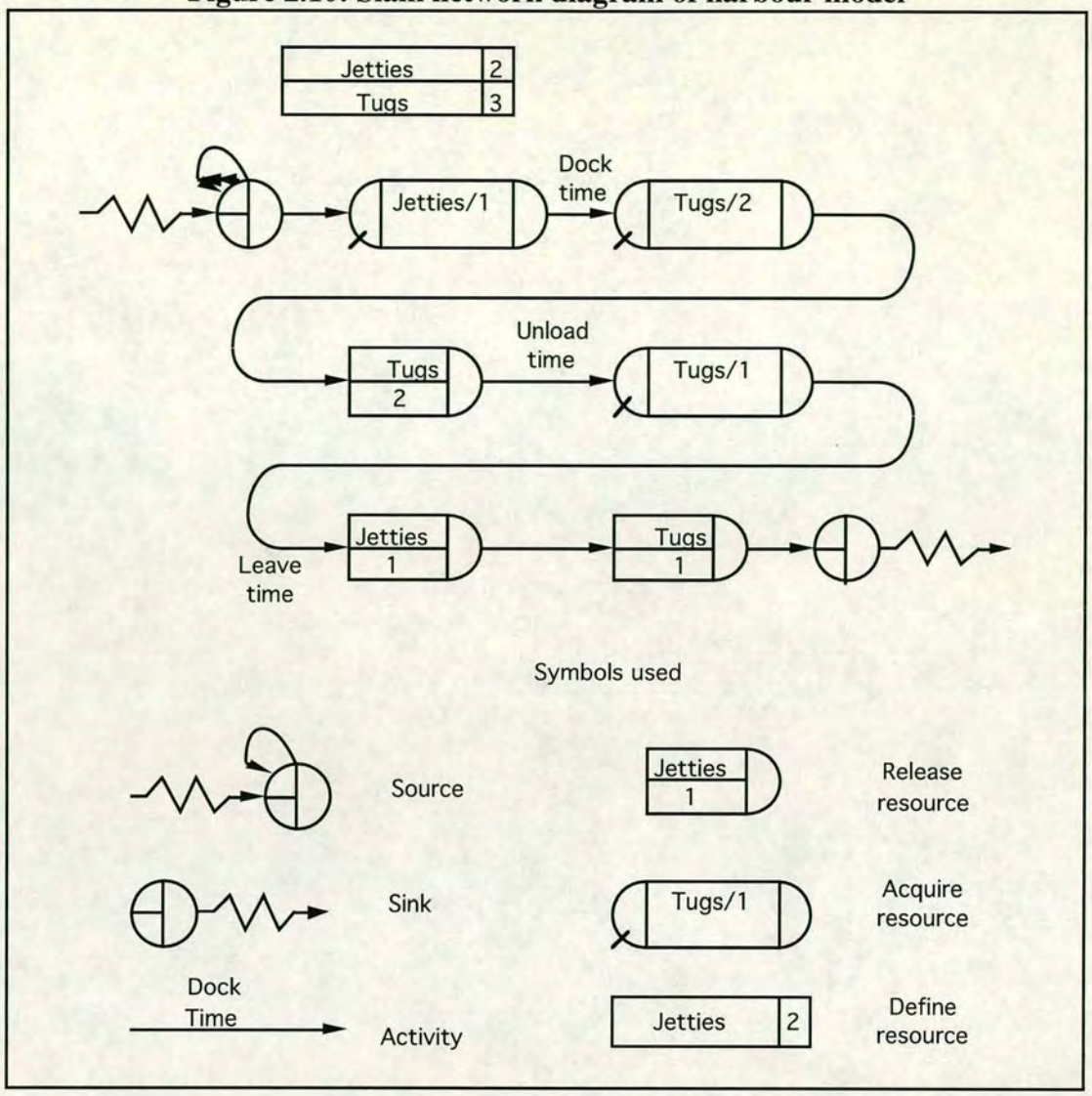
It can be seen that the resources are shown explicitly, although their use is indicated by the occurrence of their names in AWAIT nodes and FREE nodes, not by links. This is very similar to GPSS.

In figure 2.10 ships are being generated at the left and passing through queues, where they wait for servers, represented by directed edges. Each server has a delay defined, like the GPSS Advance block. By the mechanism of Files associated with each queue, different queueing disciplines can be enforced. When resources are required the ships enter AWAIT nodes until sufficient are available.

In fact the SLAM II diagrams are very similar in this simple use to the GPSS equivalent. Like them they involve lots of nodes. The full vocabulary of SLAM II network diagrams is very rich and allows expression of quite complex models. The explicit representation of resources helps readability, but the lack of graphical links from them to AWAIT and FREE nodes reduces this benefit. There is a fairly complete set of statistical collection symbols as well.

In general these diagrams work quite well for modellers who are aiming at SLAM II as a programming language. The TESS graphical input front end [94] uses the completeness of the representation to allow model generation directly from them. The problems are the explosion of detail, the orientation towards SLAM II (although this may be more apparent than real) and the lack of any convenient means of modularisation such as hierarchical processes.

Figure 2.10: Slam network diagram of harbour model



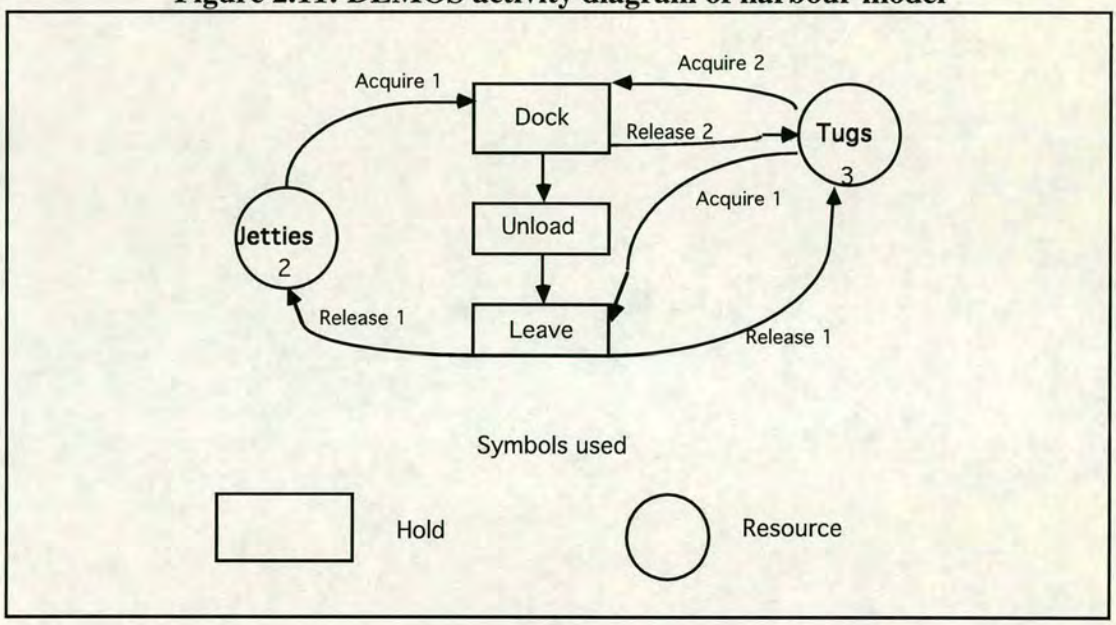
The approach is in general process based, but overlaps at times with the activity style. Thus, there are synchronisations called ACCUMULATE - which blocks process instances until a required number have reached the same state - and MATCH - which blocks process instances until attribute values can be matched between them and other instances. These illustrate an important semantic confusion which surrounds the word *process* in such discussions. In the view of Franta [27] and Birtwistle [13] it seems that a process is an instance of what is called an entity in the activity style, with all its actions encapsulated. Thus, the process based view starts from the life cycle of entities and synchronises these through external queues of various kinds, such as resources and wait until states. SLAM II, like GPSS and most other supposedly process oriented modelling systems, keeps the distinction between

a process description and the entities flowing through it, which is closer to activity style descriptions. It is in contradiction to the definition of processes used here that ACCUMULATE and MATCH are part of the process rather than an interaction with a mechanism external to it. The approach in Chapters 3 and 4 depends on this separation of process and environment.

2.4.12 DEMOS activity diagrams

The diagram in figure 2.11 is based on figure 3.5 of [13]. This includes the standard symbols of a rectangular box for a delay, annotated with a description of the associated activity, and a circle for a resource, annotated with a description of the resource and the initial amount available.

Figure 2.11: DEMOS activity diagram of harbour model



This approach seeks to merge the simplicity of activity cycle diagram with the descriptive power of a GPSS-like flowchart process description. As used by Birtwistle it gives an incomplete definition of the model and ignores many more complex possibilities. It is not even capable of expressing all of the power of the DEMOS simulation system itself. It has, nevertheless, proved popular and influential. Birtwistle himself notes that experts in the properties of the system being modelled have found it relatively easy to understand such descriptions and it has proved useful in a number of modelling exercises of different types, including computer hardware, communications protocols and factory systems. Hughes

extended the range of symbols to model interrupts and conditional waits [40]. Chapter 4 develops activity diagrams as a standard way of representing process based models.

2.5 Sub-models and hierarchies

Most modelling tools and languages started with a flat view and offer little support for sub-models. Thus SLAM's network models are an inherently flat description of a total system. The notion of hierarchy is essential, however, to the complexity and scale of many models, as well as sometimes allowing more efficient solution. Existing approaches to hierarchical modelling for either of these purposes are considered below.

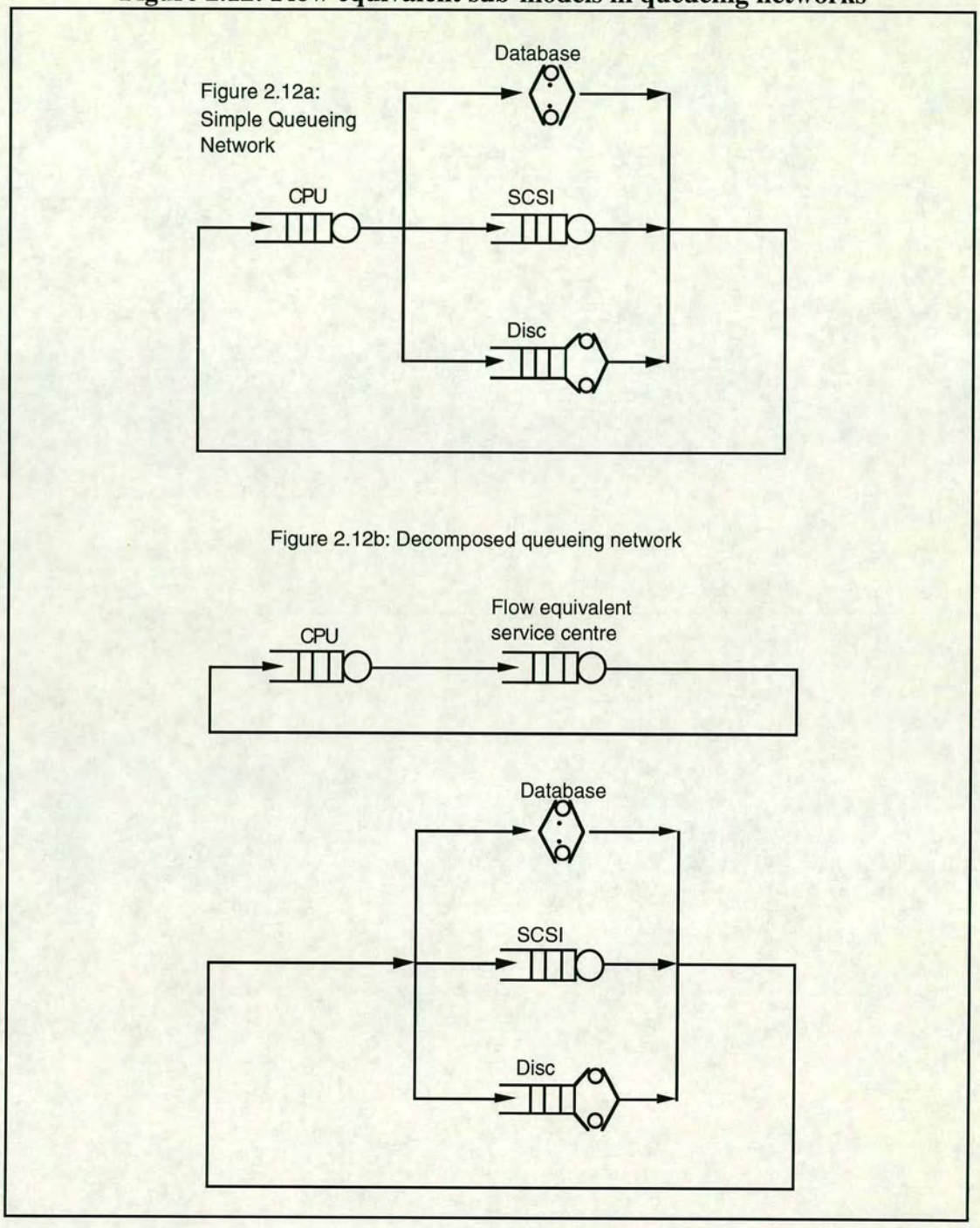
2.5.1 IPG sub-models

One exception already noted is the use of sub-model nodes in PAWS and SES Workbench IPGs. These are intended to support information hiding and reuse of previously defined model sections. There is, however, no attempt to exploit them to enhance model solution. They are allowed only a single entry and exit point and leave the underlying model flat. There is no reason that they could not form the basis of some sort of hierarchical modelling, along the lines of flow equivalent service centres, which are described next.

2.5.2 Flow equivalent service centres and other aggregated sub-models

Within queueing networks, the idea of *flow equivalent service centres* was introduced to allow pre-solution of sub-networks and the substitution of tables representing aggregate behaviour into the resulting main model. This notion of decomposition and aggregation has been generalised to a basis for heterogeneous modelling by Beilner[9] and Buchholz [17]. Combined modelling using simulations in DEMOS, generated by the PIT model editor [6], and Petri nets in GreatSPN, generated by the PNT graphical editor, were shown to be possible, using the Edinburgh Experimenter within the IMSE framework [17, 36]. PIT added a flow equivalent server node to standard DEMOS activity diagrams to support this.

Figure 2.12: Flow equivalent sub-models in queueing networks



2.5.3 Sub-models in DEMOS

DEMOS is based on SIMULA, which is object oriented, and so views ENTITYs and the synchronisation components in models, such as resources, as objects defined by classes. This allows processes to be built from collections of other objects, i.e. fully general component based modelling is possible. This is explored in Chapters 3 and 4

in the context of a model of X.25 in [69] using experience from [67] and [108]. Although DEMOS gives the best support to such sub-model structuring of any of the systems reviewed, it is not formalised within the DEMOS package or the notation of (extended) activity diagrams. The PIT tool showed the strength of the approach, by using it to add two new nodes to activity diagrams, the FESC described above and the *server*, which was an abstraction of a resource and a process imparting a delay to form something very like the service centre and associated queue of a queueing network.

Hierarchies in a modified form of DEMOS and their formalisation, represent one of the main contributions of this dissertation.

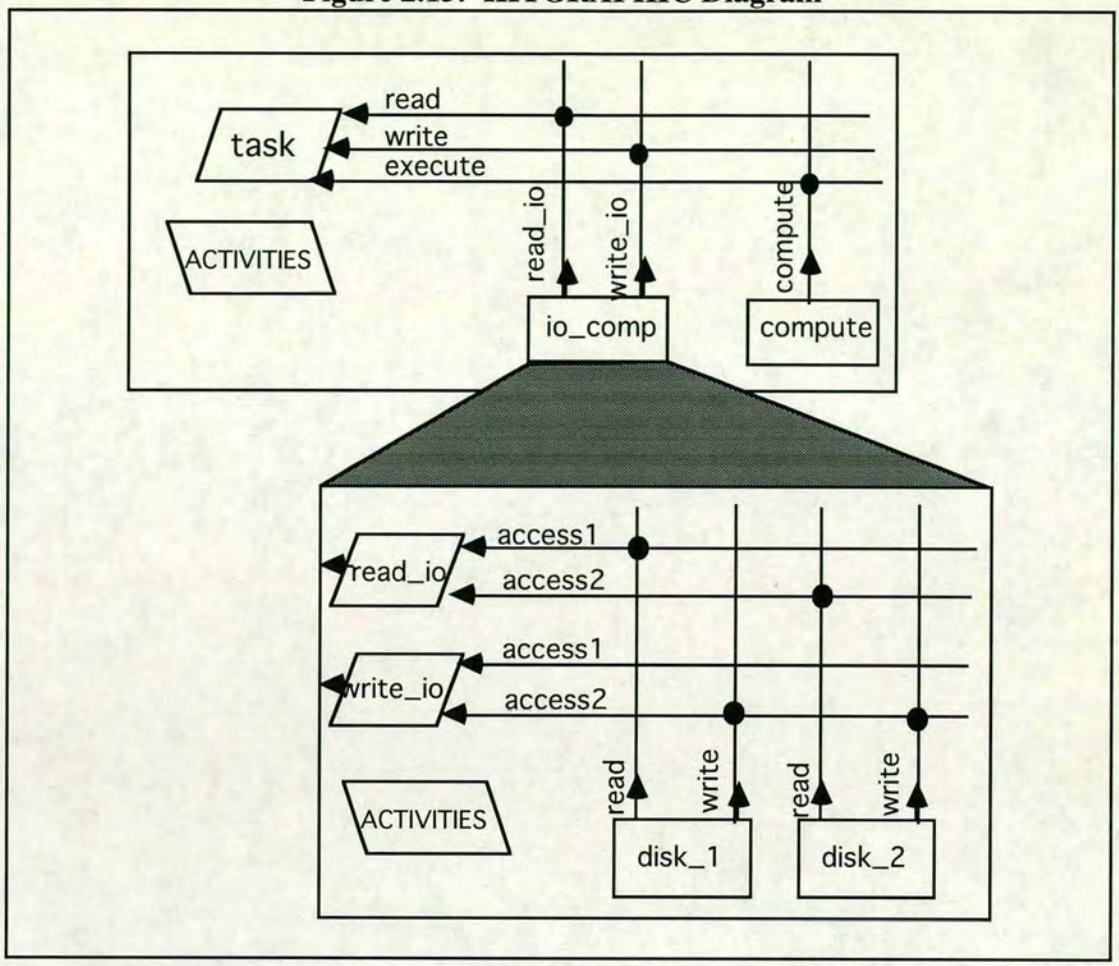
2.5.4 HIT

HIT [8,10] is specifically built to support modelling of computer systems in a hierarchical manner, based on a layered machine view of such systems. This allows modularisation, corresponding to components within layers of the real system. Such a view gives a form of description which is very natural for the types of systems considered. The user interface uses either a textual language, HI-SLANG, or a graphical model construction interface, HITGRAPHIC. In either, modules at a higher level are use services from modules lower down. At the lowest level simple services are described. Modules, termed COMPONENTS, are described as LOADs applied to MACHINEs.

Written entirely in SIMULA, HIT can generate automatically models for solution by discrete event simulation, exact solution as product form networks and approximate solution for other classes of network, numerical solution of underlying Markov chains and structured decomposition and aggregation of large models for efficient solution. HIT runs on most platforms supporting a SIMULA compiler, including most UNIX workstations and IBM and Siemens mainframes. HITGRAPHIC is written in C and runs on top of X Windows. It was developed at Universität Dortmund with initial support from Nixdorf Computer AG and BMFT.

With its combination of solvers and its hierarchical approach, HIT shows the feasibility of a general approach to description of performance models. It lacks, however, any formal behavioural semantics.

Figure 2.13: HITGRAPHIC Diagram



2.6 Formal representation of discrete event simulations

There have been a number of attempts at formalising various aspects of discrete event simulation modelling. Some have been oriented towards the modelling process, others have concentrated on models themselves.

2.6.1 Formalising the modelling process

The process of modelling is really a branch of experimental method. There are probably two major attempts which have been made to structure this.

The Conical Methodology

In the Conical Methodology [64], the software engineering lifecyle is modified to describe the simulation modelling process. In particular, the spiral model of software engineering is used as the basis of a conical model of simulation modelling.

Multifaceted Modelling

The idea that no one model can express all aspects of the system being represented is well known, but it was first formalised by Zeigler [110,111,112]. Here a systems theoretic approach is developed for simulation. Essentially Zeigler notes that the *system* corresponds to a *base model*. Such a model is unrealisable, as the level of detail required is beyond the capability of our modelling techniques. *Experimental frames* are introduced to define sets of conditions under which observations are possible and *lumped models* are models capable of solution under the conditions of one experimental frame. Computation is the means of extracting the results from a lumped model under the conditions of an experimental frame. This approach allows a hierarchical modelling framework to be developed, with higher level, more abstract models deriving some of their detailed information from lower level, detailed models of more restricted parts of the system. This notion of hierarchy is based on information flow and representation.

Using this framework, Zeigler went on to develop the DEVS formalism, described below. This framework was also a major influence on the work of he SIMMER Alvey project [41, 42, 71] and the IMSE ESPRIT II project [75].

2.6.2 Formalising simulation models

The use of simulation models poses problems in formal understanding of their behaviour at all steps in their use. Firstly, during model construction it may be desirable to use pre-existing component sub-models and to simplify the behaviour of sub-systems while preserving behavioural properties. Secondly, at the stage of verification it is important to establish that the model being used reproduces the expected behaviour of the system being modelled. Then at the stage of validation, it is important to understand the context within which the model is expected to behave in the required way and to quantify its behaviour. Finally at the stage of model solution, it is important to be able to simplify and re-use sub-models without loss of important aspects of behaviour.

DEVS

The Discrete Event System Specification formalism is a framework for describing simulation models, consistent with Zeigler's multi-faceted modelling approach [96]. Within it a model is defined by the structure:

$$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

Where:

X	is the set of external input event types;
S	is the sequential state set;
Y	is the set of output events controlled by M;
δ_{int}	is the internal transition function defining state transitions due to internal events;
δ_{ext}	is the external transition function defining state transitions due to input events;
λ	is the output function;
τ	is the time advance function.

DEVS recognises two types of model, *atomic* and *coupled.* An atomic model is complete and does not depend on any other models for its execution. A coupled model is connected to other models via input and output events. Models are defined to be closed under coupling, so that from an external viewpoint there is no difference between these two categories. A model is defined through its input and output interfaces.

Thus DEVS supports an hierarchical modelling concept, based on information flow. This is rather different from the view of hierarchical modelling that is developed in this dissertation, since the notion of coupling used will be in terms strictly of process interactions, which implies the type of the source of inputs. In fact it is possible that the two approaches may be complementary.

Implementations of DEVS have been made in Scheme [48] and CLOS [91]. Case studies include [46, 45]. It has been applied to continuous systems modelling in [28].

Simulation nets

Simulation nets are, apart from some work with Petri nets discussed below, the most closely related formalism to that presented in this dissertation and produces some comparable results in [89] and [109]. Like the present work, the approach is to try to identify behavioural properties of simulations from the formalism.

In Schruben's original paper the concept of simulation nets is explained using a diagrammatic representation. Models consist of annotated directed graphs, where the vertices correspond to events and the edges correspond to the influence of an event on other possible events. This influence can be to schedule an event after a delay, subject to a condition, or to cancel an event after a delay, subject to a condition. The example in figure 2.7 represented the simple harbour model as a simulation net. Using Schruben's event reduction rules, the events *end docking* and *start unloading* have been combined. This corresponds to the fact that a release of a resource can never be blocking.

In their later paper Yücesan and Schruben investigate further the use of simulation nets to express behavioural properties of models. They focus on the structure of the nets in what they now call *Simulation Graph Models*. This uses graph isomorphism and has no idea of observation equivalence or bisimulation.

This later work introduces the notion of *parameterised vertices* and *edges*. Each vertex is allowed to have a set of state variables which are bound to a set of corresponding expressions associated with an incoming edge. In the usual graph theoretic notation, simulation graphs are defined. A graph G, as a triple of ($V(G)$, $E(G)$, Ψ_G), where $V(G)$ is the set of vertices, $E(G)$ is the set of edges and Ψ_G is the incidence function associating each edge with an ordered pair of vertices. A simulation graph is a quadruple of ($V(G)$, $E_S(G)$, $E_C(G)$, Ψ_G), where the edges are divided into scheduling and cancelling ones.

The annotation of such nets consists of:

$\mathcal{F}$ = $\{ f_v : \text{STATES} \rightarrow \text{STATES} \mid v \in V(G) \}$,
the set of *state transition functions*;

$\mathcal{C}$ = $\{ C_e : \text{STATES} \rightarrow \{0,1\} \mid e \in E_s(G) \bigcup E_C(G) \}$,
the set of *edge conditions*;

$\mathcal{T}$ = $\{ t_e : \text{STATES} \rightarrow \Re^+ \mid e \in E_s(G) \}$,
the set of *edge delay times*;

Γ = $\{ \gamma_e : \text{STATES} \rightarrow \Re^+ \mid e \in E_s(G) \}$,
the set of *event execution priorities.*

Using results from Schruben's earlier paper, which are formalised and revised, notions of equivalence under expansion and of equivalence through isomorphism are developed.

For figure 2.7 above these have the following values:

$\mathcal{F}$ = $\{ f_i \mid i = 0..7 \}$ = { Jetties := 2, Tugs := 3, in := 1;
ia := ia+1; Jetties := Jetties - 1;
Tugs := Tugs - 2 ;
Tugs := Tugs + 2 ;
no change;
Tugs := Tugs - 1;
Tugs := Tugs + 1, Jetties := Jetties + 1
}

$\mathcal{C}$ = $\{ C_{1,2}; C_{2,3}; C_{5,6} \}$ = { Tugs>=2; Jetties>=1; Tugs>=1 }

$\mathcal{T}$ = $\{ t_{1,1}; t_{3,4}; t_{4,5}; t_{6,7} \}$ = { t_{arrive}; t_{dock}; t_{unload}; t_{depart} }

Γ = $\{ \gamma_{0,1}; \gamma_{1,1}; \gamma_{1,2}; \gamma_{2,3}; \gamma_{3,4}; \gamma_{4,5}; \gamma_{5,6}; \gamma_{6,7} \} = \{1, 1, 1, 1, 4, 1, 2, 3\}$

In [89] rules 1 and 2 of event graph analysis are of no relevance to the work of this dissertation. However, in section 3.3, rule 3 states that: *Event scheduling priorities are required when the intersection of the state variable sets of two vertices is non-empty.*

Events 2, 4, 6 and 7 require relative event scheduling priorities since these events share the state variable Tugs. Events 3 and 7 share variable Jetties. We choose to give higher priorities based on quantities released to resources and to closeness of subsequent release after acquires.

The model presented in figure 2.7 is fully detailed and can be simplified by the event reduction rules in section 4 of [109], which formalise and correct those in [89].

Rule 4a: *Equivalent SGNs are possible with and without an event vertex k, if vertex k has no conditional exiting edges and if all edges entering vertex k have zero delay times. If rule 4a applies vertex k may be combined with the originating vertices of its entering edges. State variable changes are added to those in these preceding vertices. k must have a higher scheduling priority than any of these preceding vertices.*

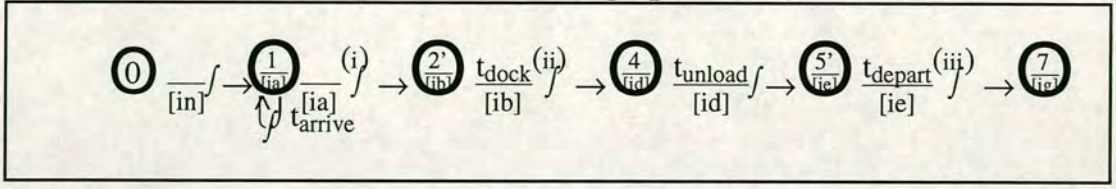
Rule 4b: *Equivalent SGNs are possible with and without an event vertex k, if vertex k has no conditional exiting edges and if there are no state variable changes associated with it.*

Rule 4c: *Equivalent SGNs are possible with and without an event vertex k, if vertex k has no conditional exiting edges and if all edges exiting vertex k have zero delay times. If rule 4c applies vertex k may be combined with the termination event vertices of its exiting edges. State variable changes in k are added to those in the succeeding vertices. k must have a lower scheduling priority than any of these preceding vertices.*

Rule 5: *If $f_k = \varnothing$ for all interior vertices k of an unconditional event tree, then only the leaf vertices of the tree need be included in a simulation graph.*

Rule 4a allows us to remove vertices 3 and 6 in the harbour model, giving the revised model in figure 2.14 below.

Figure 2.14: Simplified simulation graph of the harbour model



Rules 4b, 4c and 5 cannot be applied in this model and so no further event reduction is possible.

If the model is modified to allow potential deadlock, by reversing the order of vertices 1 and 2 in the original, Schruben's rule 3 correctly identifies a possible problem, but does not show exactly what form it might take. In assigning the event scheduling priorities the modeller has to be alert to resolving deadlocks along with a number of other so called *simultaneous event problems*.

In Chapter 6 the use of Schruben and Yücesan's notions of equivalence is discussed further and compared with the approach developed in this dissertation. In general they are less powerful, since they require a stronger notion of equivalence than the observation equivalence used by CCS.

Petri nets

As discussed above, Petri nets offer a powerful, but verbose, graphical formalism for the description of systems, especially those with genuine concurrent behaviour. Prior to their extension for performance modelling, culminating in GSPNs and DSPNs, they were used for structural analysis of systems, to determine possible deadlocks, livelocks etc. This is very close to the efforts made in this dissertation with respect to process algebras. Most tools built for Petri net modelling incorporate algorithms for detection of deadlocks, traps and invariants.

When the use of Petri nets for describing simulation models was introduced, it was quickly realised that these properties could be useful. It was also realised that it might be possible to eliminate redundant states and to ensure coverage of the state space. This depends on the generation of the *reachability graph*, which enumerates all states, as combinations of numbers of tokens in places (*markings*), and the possible transitions between pairs of states.

A major weakness of Petri nets is their lack of compositionality, so that it is very difficult to identify equivalent states or sub-nets. When sub-nets are combined, there is no guarantee that properties of the sub-nets will be preserved. The reachability graph depends on both the structure of the net and on the initial marking of the net.

2.7 Process algebras

Process algebras have been developed for similar purposes to Petri nets, i.e. as a means of representing the behaviour of concurrent systems with communication between components. The two best known examples are probably Hoare's

Communicating Sequential Processes (CSP) [38] and Milner's Calculus of Communicating Systems (CCS) [58].

Compared to Petri nets, process algebras are algebraic rather than graphical, allow preservation of properties of components when composing larger models from components and, often, use an interleaving semantic model rather than a fully concurrent one. While the first of these may seem to argue against them for the purposes of this dissertation, the other two make them extremely likely candidates for representing discrete event simulation models. The lack of a widely used graphical notation is not seen as important, since it proves relatively straightforward to generate the algebraic form from the graphical notation for process based simulation given in Chapter 4.

2.7.1 CCS

The Calculus of Communicating Systems forms the core of the formal semantics for process based simulation developed in Chapter 3. It was created to model the behaviour of systems which can be described in terms of communicating *agents*. Consider first the basic calculus [58]. This contains the following primitives for defining agents, which will be used in later chapters:

sequential composition	$a.B$	after action a the agent becomes a B
parallel composition	$A \mid B$	agents A and B proceed in parallel
choice	$A + B$	the agent behaves as either A or B, but not both, depending on which acts first
restriction	$A \setminus M$	the set of labels M is hidden from outside agents
relabelling	$A[a_1/a_0,..]$	in this agent label a_1 is renamed a_0
the null agent	0	this agent cannot act (deadlock)
the divergent agent	$\perp$	this agent can cycle indefinitely and unobservably

Here identifiers starting with lower case letters denote labels which represent complementary action pairs, where the use of a label with, $\bar{c}$, and without, c, an

overbar distinguishes two halves (output and input) of an action, both of which must be possible before it can proceed. Identifiers which begin with an upper case letter define agents. Agents are constructed from the forms given above.

Symbolic names for agents are defined using the infix binding symbol, $\stackrel{\text{def}}{=}$.

In the Concurrency Workbench this operator is replaced by the prefix operator `bi`.

Thus the equations

$$A \stackrel{\text{def}}{=} b.C$$

and

```
bi A b.C
```

are equivalent in the two forms.

CCS uses a notion of *observation equivalence*, which depends on the assumption that two agents are equivalent if any differences in their behaviours cannot be distinguished by an observer. Where two agents containing the two sides of a complementary action are combined in parallel, the resulting agent may hide the action and regard it as internal. CCS calls such internal actions τs. Under many circumstances such internal actions have no effect on the observable behaviour of agents and so may be ignored. This is not always so, however, notably when a τ is the prefixing action of one half of a choice.

CCS is essentially a labelled transition system, where each combination of agents can be thought of as one state and each communication action labels the transition between one state and its successor, in a similar way to markings in Petri net defining states. The semantics of CCS are defined using Plotkin style operational semantics expressed as inference rules on labelled transitions.

Transitions are of the forms:

$A \xrightarrow{tugAcq2} B$

A simple transition, where A engages in one side of the action $tugAcq_2$ and evolves into agent B. Instead of a single action, a sequence of actions can be used to label such a transition.

$A \stackrel{tugAcq2}{\Rightarrow} B$

A transition which abstracts from silent actions. Thus, any number of ts can be allowed to precede and succeed *tugAcq2*. Where the label is a sequence this generalises the abstraction accordingly.

By using the notion of bisimulation as its basis of equivalence, CCS is able to detect equivalence for a wider class of models than the use of isomorphism would permit. It is also inherently compositional, allowing bisimulation results proved for components to be preserved by its combinators and so reducing the effort of proving properties of larger models constructed in this way. This will prove vital in establishing the semantics of hierarchical models in Chapter 3.

Deadlock occurs if none of the outstanding actions at a certain point is matched by its complement in another agent with which it is composed in parallel. In strict terms, this also requires the action to be restricted from outside the system, otherwise an undefined agent might still activate it.

2.7.2 Temporal CCS

Temporal CCS [98,60] is an extension to Milner's basic CCS, which allows both explicit delays and wait for synchronisation (asynchronous waiting), in a manner superficially strikingly similar to DEMOS. It adds the primitives:

fixed time delay	(t)
wait for synchronisation	δ
non-temporal deadlock	$\underline{0}$

The deadlock now extends to cover situations where time cannot pass, since all parallel components must be ready to advance time for it to move on. Put another way, if there are components composed in parallel where some have as their current

action an unsatisfied complementary action, and other agents have a time delay, the system is in temporal deadlock. Non-temporal deadlock allows indefinite idling, i.e. all processes are able to wait indefinitely for actions which cannot happen and so they cannot evolve.

The wait for delay is sometimes written by underlining the next action. In the Concurrency workbench it is written as a $ symbol preceding the next action.

2.7.3 Synchronous CCS (SCCS)

An earlier variant of CCS is Synchronous CCS (SCCS) [58]. This offers greater realism in describing synchronisation of genuinely concurrent systems, but is not really suited to the purposes of this work. In this dissertation the problem is to represent the interleaving behaviour of a simulation language, not the behaviour of the systems it models. As will be seen in Chapter 6, there are real problems in using a sequential language to model truly concurrent behaviour, but this is not addressed by pretending that the language is genuinely parallel.

2.7.4 Concurrency workbench

The Concurrency Workbench (CWB) [20,61] is a tool that automates the checking of assertions about CCS models in order to establish properties the systems they describe. It supports the basic calculus, the temporal extension to this and the synchronous variant. The CWB allows testing of expressions in the modal μ-calculus, which is a process logic (see 2.8.2 below).

The CWB is used in Chapters 3 and 6 to evaluate the possibility of automating some kinds of reasoning about process based models. The possibility of generating CCS models suitable for use with the CWB automatically from activity diagrams, along with their DEMOS equivalents, is described in Chapter 5.

2.7.5 Stochastic extensions to CCS

A number of attempts have been made to add stochastic behaviour to CCS, or to similar algebras, including Jou and Smolka [47], Larsen and Skou [53] and Tofts [99]. Some of these have been concerned with un-timed behaviour, where choices have branching probabilities or weights attached to them. This sort of model can be used to think about reliability and limited notions of timing. In some cases the notion

of bisimulation is made into a probabilistic concept, such that two systems are bisimilar if the probability of different behaviour is less than some defined threshold.

TIPP [31] and PEPA [37,30] are examples of CCS influenced process algebras defined with the express purpose of representing models solvable for performance measures. Thus they both allow stochastic behaviour in terms of both times and branching probabilities. These calculi are designed for numerical solution of models, in the same way that GSPNs and DSPNs have been developed. They are obviously capable of being simulated and may be of some help in answering some of the open issues of this dissertation.

2.7.6 Other work using process algebras to express simulation semantics

Three other pieces of work have been reported where process algebras have been used to express properties of discrete event simulation models.

Strulo [96] defined a version of CCS whose semantics described Generalised Semi-Markov Processes. It is known that it is possible to use GSMPs to describe many simulation models and so the claim was made that this calculus could be used to formalise real simulation languages. Although Strulo relates some of his results to systems like DEMOS, the end result is still totally theoretical and he never solves or executes any models derived from his descriptions. Nor does he show that useful behavioural properties can be derived.

In an unpublished technical report [100], Tofts used the Synchronous Calculus of Communicating Systems (SCCS) to explore some of the basic mechanisms of DEMOS. Although this work duplicates some of that presented here, it post-dates it and covers a restricted part of the problem, with assumptions about the behaviour of DEMOS which are not always valid. This and Strulo's work assume that the problem is to examine the world that simulation models purport to represent, rather than the capabilities of a simulation language.

Work presented by the author in a joint paper with Tofts and Birtwistle [16] offers a partial representation of non-hierarchic process based simulation behaviour using basic CCS. This was published jointly in recognition of the independent realisation by the three authors of the possibilities of the approach. Again the work covers only

a part of what is presented here, assuming an idealised sub-set of the facilities in DEMOS.

2.8 Process logics

If process algebras represent a useful way of describing models, with a formally defined semantics, it is natural to use a corresponding process logic to frame properties and queries concerning these models. Although the Concurrency Workbench, for instance, allows simple properties, such as the presence of deadlock, to be queried directly, it needs a suitable logic to express more specific properties and questions. Formally such logics are known as *modal logics* and express assertions about changing state. Such logics are not confined to reasoning about CCS. They apply generally to labelled transition systems.

There is an appealingly simple modal logic, known as Hennesy-Milner logic [32], for expressing assertions about the immediate possibilities for a model. There is also an extended modal logic, with fixed point operators allowing the expression of recursive definitions, known as the modal μ-calculus. Within the CWB, the modal μ-calculus [95] is used for this purpose.

2.8.1 Hennesy-Milner logic

The description here follows the outline of Milner's presentation in [58].

Consider the simple system

$$S_1 \stackrel{\text{def}}{=} a.S_2$$

$$S_2 \stackrel{\text{def}}{=} a.S_3$$

$$S_3 \stackrel{\text{def}}{=} b.S_3$$

Using Hennesy-Milner logic it is possible to assert properties of a system's states, using the following operators:

- *satisfaction* $\models$ - the agent on the left hand side of the operator satisfies the formula on its right hand side.

- *possibility* <> e.g. it is possible to make an *a* move both from S_1 and from *S2*.

These are expressed respectively as:

$S_1 \quad \models \quad$ <a> true

and $S_2 \quad \models \quad$ <a> true

The state true implies unconditional satisfaction. It is shorthand for the empty conjunction,

$$\bigwedge_{i \in \varnothing} \mathcal{F}_i.$$

- *non-satisfaction* $\not\models$ e.g. S_3 cannot make an *a* move, i.e.

$S_3 \quad \not\models \quad$ <a> true

which means $S_3 \quad \models \quad \neg$<a> true

It is possible to distinguish between S_1 and S_2 if from S_1 if it is possible to make one *a* move followed by another, but not to do this from S_2. This is expressed as:

$S_1 \quad \models$ <a> <a> true

and $S_2 \quad \not\models$ <a> <a> true

- *necessity* [a] - the dual operator to <a>. If:

$S_1 \quad \models \quad$ [a] $\mathcal{F}$

then by performing the move *a*, S_1 must always reach a state where $\mathcal{F}$ holds.

<a> requires at least one of its currently possible *a* moves to reach the following state; [a] requires all of its currently possible *a* moves to reach the following state.

Some useful extra notation:

	–	stands for all actions;	
	–*k,l,m*	stands for all actions except *k,l,m*;	
	<a,b,c>S	is short for	<a> S ∨ <b> S ∨ <c> S;
and	[a,b,c]S	is short for	[a] S ∧ [b] S ∧ [c] S.

There are also weak forms of the possibility and necessity operators, which disregard any τs:

- weak possibility ≪a≫

  Weak possibility can be defined as follows:

  $$E \models \ll\gg\Phi \quad \text{iff} \quad \exists F \in \{E' \mid E \stackrel{\varepsilon}{\Rightarrow} E'\} \,.\, F \models \Phi$$

  I.e. E can silently evolve into a process satisfying Φ. Hence:

  $$E \models \ll a\gg\Phi \stackrel{\text{def}}{=} \ll\gg\langle a\rangle\ll\gg\Phi$$

- weak necessity [[a]]

  Weak necessity can be defined as follows:

  $$E \models [\![\,]\!]\Phi \quad \text{iff} \quad \forall F \in \{E' \mid E \stackrel{\varepsilon}{\Rightarrow} E'\} \,.\, F \models \Phi$$

  I.e. E cannot silently evolve into a process failing Φ. Hence:

  $$E \models [\![a]\!]\Phi \stackrel{\text{def}}{=} [\![\,]\!]\,[a]\,[\![\,]\!]\,\Phi$$

Here are some common uses of Hennesy-Milner logic:

E	⊨	$[a]$ F	E cannot make an a move
E	⊨	$\langle a\rangle$ T	E can make an a move
E	⊨	[–] F	E is deadlocked
E	⊨	<–> T	E can make a move of some sort
E	⊨	<–> T ∧ $[-a]$ F	E can only make an a move

2.8.2 The modal μ-calculus

Hennesy-Milner logic is good for asking questions one or two moves ahead, but cannot cope with recursive definitions. By adding just one construct - fixed point operators - to Hennesy-Milner logic, the result is the modal μ-calculus. This is in effect a powerful temporal logic, allowing one to express notions of eventuality and invariance of states and actions. Although the modal μ-calculus is much more general than even a process logic, the discussion here is restricted to its use with CCS.

More complete, fairly readable accounts of the modal μ-calculus can be found in Stirling [95] and Aldwinckle, Nagarajan and Birtwistle [3]. A useful introduction to the representation of temporal logics in the modal μ-calculus is given in Dam [24].

A fixed point equation might have the form:

$$Y \stackrel{\text{def}}{=} \langle a\rangle\langle b\rangle\, Y$$

meaning that each state in Y has the property of being able to perform an *a* action followed by a *b* action and then reaching a state in the original set, Y. Once we have allowed such recursive definitions we can examine the properties of fixed point equations and find sets of states which satisfy them, within agents. Not all such equations have solutions, nor are their solutions guaranteed to be unique. However, a restriction that there must be an even number of negations prefixing a recursively defined variable in an equation guarantees that there must be at least one solution. Formally, this property defines that the equation is *monotonic*.

It is worth noting that a property with respect to a model defines the set of states where that property holds, i.e. the property and the set of states are different ways of expressing the same thing.

There are two very important fixed point operators, defining the *maximum* and *minimum* fixed points of a recursive equation. The maximum fixed point is related to the fact that the union of any two solutions to a fixed point equation is a subset of a further solution. This superset is the closure under deduction of the union of the two initial sets. The maximum fixed point of an equation is the closure under deduction of the union of all fixed points of that equation, i.e. it contains every state which can form part of a solution. The minimum fixed point is related to the fact that the intersection of any pair of solutions contains a solution. Thus the minimum fixed point of an equation is the smallest solution to that equation and is a subset of the intersection of all fixed points. It contains only those states guaranteed to be in every solution. It is often the empty set.

Whilst it is not always obvious how to interpret fixed point modal formulae, the general idea is that a maximum fixed point expresses some property which *always* holds (an invariant), while a minimum fixed point expresses a property which will *eventually* hold. When verifying systems maximum fixed points are useful for expressing safety properties and minimum fixed points for expressing liveness properties.

Some examples yield to intuition. For example, following [3]:

$$Y \stackrel{def}{=} (\langle x \rangle T \vee [-]Y)$$

has a minimum fixed point which can be read as saying that it is possible to perform an *x* action or all actions lead to a situation where it is eventually possible to do so. The maximum fixed point of the same equation denotes the set of all states.

As another example, the equation:

$$Y \stackrel{def}{=} (\langle x \rangle T \vee \langle - \rangle Y)$$

has a minimum fixed point meaning that it is possible to perform an *x* or there is a derivative leading to such a possibility. Its maximum fixed point denotes all states capable of an *x* action or of performing some infinite sequence of actions.

Notation for fixed points

The least fixed point is conventionally written in the form:

$$\mu Z.\langle a \rangle Z$$

meaning the least fixed point solution of

$$Z \stackrel{def}{=} \langle a \rangle Z$$

Similarly

$$\nu Z.\langle a \rangle Z$$

is the maximum fixed point solution of the same equation.

Within the concurrency workbench, these are expressed as

min(Z.<a>Z) and max(Z.<a>Z), respectively.

The workbench also uses & for the logical connective *and*, ∧, and | for the logical connective *or*, ∨.

Some useful intuitive interpretations

The modal μ-calculus can be used to express many more conventional temporal logic operations. Since these tend to be more intuitive, Birtwistle has defined some within the Concurrency Workbench, using macro definition capabilities which support such definitions. Some examples follow.

Box, Weak Until and Strong Until are taken from the Concurrency Workbench technical note [61], others are based on examples in [3] and in [95].

- **Box**: $S \models \text{Box } \Phi$ or $S \models [] \Phi$

  is true if Φ holds in each state reachable from S.

  E.g. the test for whether S can deadlock is simply S⊨ *BOX* <–> true (we ask of each state reachable from S "can you make a move?").

  max(X.P & [-]X)

is the branching time temporal logic operator which says that P holds of an agent and continues to hold recursively for all derivations.

- **WeakUntil**: $S \models \text{WeakUntil } \Phi\ \Theta$

is true if Φ holds for all derivations until a state is reached where Θ holds. This is weak, since Θ need never hold for the property to be true.

It can be written in the concurrency workbench as:

max(X.Q | (P & [-]X))

- **StrongUntil**: $S \models \text{StrongUntil } \Phi\ \Theta$

is true if Φ holds for all derivations until a state is reached where Θ holds. This is strong, since Θ must eventually hold for the property to be true.

It can be written in the concurrency workbench as:

min(X.Q | (P & [-]X | <->T))

- **Poss**: $S \models \text{Poss } \Phi$

is true if S or (at least) one state reachable from S satisfies Φ.

It can be written in the concurrency workbench as:

min(Y.P | <->Y)

- **Event**: $S \models \text{Event } \Phi$

is true if Φ holds for (at least) one state on each and every path from S.

It can be written in the concurrency workbench as:

min(Y.P | (<->T & [-]Y))

- **Can**: $S \models \text{Can } \Phi$

is true if Φ holds along at least one path from S.

It can be written in the concurrency workbench as:

min(Y.P | <->Y)

- **Loop**: $S \models \text{Loop } \Phi$

is true if there is an unending path of Φ states from S.

E.g. $POSS(LOOP \langle\tau\rangle \text{true})$ is a test for livelock.

It can be written in the concurrency workbench as:
max(Y.P & (<->Y))

- **Must**: $S \models \text{Must } p$

  is true if the only move that S can make is a p move.

It can be written in the concurrency workbench as:

[-p]F

- **Nec**: $S \models \text{NEC } p\ q$

  is true if, however the system evolves, we cannot do a q until after a p

It can be written in the concurrency workbench as:

max(X.(<p>T | [-]X) & [q]F)

This is based on the weak until operator above, substituting the inability to perform a q for Q and the necessity of performing of a p for P. This could also be expressed in the corresponding strong form if required.

- **Cycle$_n$**: $S \models \text{Cycle}_n\ p_1 \ldots p_n$

  is only true if, however the system evolves from S, $p_1 \prec p_2 \ldots \prec p_n \prec p_1 \ldots$ where $\prec$ reads *must come before*. This is a useful test to check that agents maintain their integrity and perform actions in the expected sequence no matter what the rest of the system does.

Following [3] and using these operators on a system SYS_2

U_1	$\stackrel{\text{def}}{=}$	$n_1.gT.sc_1.ec_1.\overline{pT}\ .U_1$
U_2	$\stackrel{\text{def}}{=}$	$n_2.gT.sc_2.ec_2.\overline{pT}\ .U_2$
Sem	$\stackrel{\text{def}}{=}$	$\overline{gT}\ .pT.Sem$
SYS_2	$\stackrel{\text{def}}{=}$	$(U_1 \mid U_2 \mid Sem) \setminus \{ gT, pT \}$

the Workbench can check such assertions as:

$$SYS_2 \models BOX\,[sc_1]\,(NEC\ ec_1\ sc_1 \wedge NEC\ ec_1\ sc_2)$$

i.e. after U_1 enters its critical section $BOX[sc_1]$ (i.e. *from every state in which an* sc_1 *action is possible, do it and then*) it must exit its critical section ec_1 before re-entering its critical section via sc_1 or before U_2 is permitted access to its own critical section via sc_2.

Chapter 3

Defining simulation behaviour formally

3.1 Introduction

This chapter begins by defining carefully the concept of process interaction based simulation in English. It then defines it in terms of Milner's Calculus of Communicating Systems (CCS) [58] and shows that this is helpful in understanding the true behaviour of simulation packages, of models written using them and of the components in such models.

It then develops some requirements for modifications to the DEMOS package for completeness and to make it easier to understand the behaviour of models written using it. This analysis is used in defining the vocabulary of the graphical notation in Chapter 4. These are developed further in Chapter 5, where the modified package is implemented in terms of the graphical formalisms of Chapter 4.

3.2 Process interaction

Although the process view of simulation has a long history, its precise meaning has remained loosely defined. Even the most complete statement [27] is informal and based on a particular implementation. This chapter aims at providing a rigorous definition of such a view. *Process interaction* models systems at two levels. The basic level describes autonomous objects in terms of their behaviour. This can be represented as a finite state machine, a life history, an algorithm etc. Such objects are sequential processes in CSP [38] or agents in CCS. The higher level defines the behaviour of a system in terms of instances of such autonomous objects and of their interactions. The types of interaction allowed vary, but all are of two basic sorts, scheduling and waiting. *Interaction mechanisms* will be seen to be pairs of such interactions, linked by an object such as a resource or a queue.

In this dissertation a set of interaction mechanisms is used which relates closely to those supported in DEMOS [13], but other possibilities exist, such as SLAM [82,94], which are consistent with it. In this section these are defined from the perspective of a simulation language.

3.2.1 Interaction of processes

This section discusses the construction of processes and systems from components and sub-systems. It assumes that basic processes are defined in the general sense used above. For the purposes of hierarchical modelling all that we need to know is that in a sequential process the behaviour of an autonomous part of the system has been defined and that all points at which it interacts with other processes are visible. An interaction may actually become internal when an instance of a process is generated, if no other process shares in it, *e.g.*. a resource may be unshared in a particular model and can be disregarded.

A *process type* is the definition from which process instances are generated. It defines the behaviour, variables and interactions (through defined *synchronisation mechanisms*) of any instance. It does not define the current state of a particular instance of this type. Nor does it specify with *which* other process types or instances any particular instance of this type interacts. The state variables in a process type implicitly include a local sequence counter, which records the point in its execution that an instance generated from a type has reached.

Any *process instance* is derived from a process type by giving values for its current state and the synchronisation objects through which it shares interactions. The state of a process is the point it has currently reached in its behaviour (as defined by its local sequence counter) and the values of any explicit internal variables it possesses.

Each *interaction* within a particular process type is associated, in any instance of that type:

with one or more potential states which enable that interaction,
with a synchronisation object through which it is shared,
with zero or more processes with which it is shared.

In addition each interaction is of one of the classes defined below. Any particular interaction may involve one or more formal parameters, whose actual levels can be

constant, functions of local or of global state variables, functions of parameters of the enclosing process or functions of parameters of the environment. In practice it is sensible to restrict them to being constant or locally defined.

To create an intuitively well defined way of modelling, it seems sensible to make it match the construction of real systems. In a real world system, each visible interaction of a particular component is shared with at least one other component capable of joining in that type of interaction. When combining components to form complete systems or larger components, all such interactions are matched and, as appropriate, used to connect components. If a combination forms a new, compound component (subsystem) there may still be some unmatched or partially matched interactions. If all required connections have been made, there is a complete system. This includes its working environment, which can be viewed as a component matching any internally unsatisfied interactions. The complete system can now begin operation.

In process based modelling, the composition of sub-models and models from component instances takes place in the same way. Instances of modelled processes are combined by matching interactions until no unsatisfied interactions remain, giving a potentially executable model. At any level, state variables may be introduced into the model. These include, explicitly or implicitly, references to other component process instances at that level. Such variables are regarded as enclosed by the textual scope of the sub-model or model in which they are introduced. Access to them from outside that scope, other than for monitoring and statistical collection, is restricted to instantiation via formal parameters of the component where they are introduced or to internal actions of that component. Every variable must have a defined initial value which is a constant or a function of the parameters of that component. Once appropriate arguments and initial values for internal variables are supplied, both the model and its environment are complete and ready for execution. If some values are left free, there is a complete, but parameterised, model, suitable for use in multiple executions within an experiment.

In fact it is not always the case that all process instances are defined in a "complete" model. Often the number of identical processes entering into an interaction is left "free" as a parameter of the final model. This may also be the case where chains and rings of linked components are defined within the structure of the model. Such

models are incomplete in the sense used here and the parameterisation of these models is on two levels; one completes its structural definition and the other defines the environment for its execution.

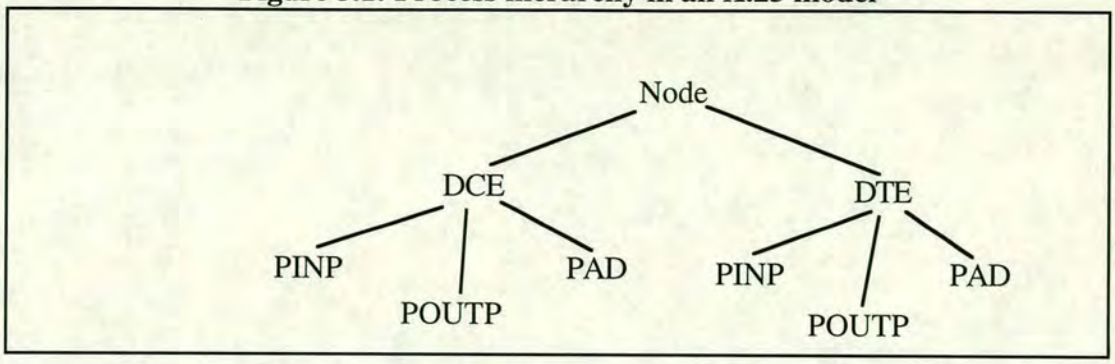
3.2.3 Hierarchies of Processes

Leaving aside the definition of internal behaviours for the moment, assume that a process is a black box with the required properties and only its interactions visible. The definition of a model introduced above says that all interactions must be correctly satisfied for it to be complete. What then of a collection of process instances with some interactions matched and some still unconnected? Such a group is made into a composite object, hiding the individual processes and any interactions among them which are completely satisfied, so long as those which are unsatisfied remain visible. The result can itself be regarded as a process. In Chapter 4 this will form the basis of the hierarchical extension to activity diagrams, called *configuration diagrams.*

The term *compound process* is introduced for such a composite and the term *atomic process* for underlying simple processes. Note that, while compound processes can be formed from any collection of processes in a model, in practice it is most useful to reflect some structure of the system being modelled, since it is unhelpful that disjoint sets of processes be included in one compound process or that closely coupled sets be split. This views a model as a tree of process instances, with a complete model as the root and atomic models as the leaves. Since the starting point in this composition of processes is any collection of processes and the outcome is a process, it is recursive and, by induction, it works for any number of levels.

Figure 3.1 shows the structure of the X.25 model reported in [69]. This model is used again as an example of graphical modelling in Chapter 4. The *Node* process represents a complete wide area network node, which connects users to the physical network and maintains virtual circuits. The two components of a *Node* are a *DCE* and a *DTE*, which are responsible for the interface to the network and to the users. Each of these contains in turn a *PINP* (Packet Input Process), a *POUTP* (Packet Output Process) and a *PAD* (Packet Assembler/Disassembler). *Nodes*, *DCEs* and *DTEs* are compound processes. *PINPs*, *POUTPs* and *PADs* are atomic processes.

Figure 3.1: Process hierarchy in an X.25 model



3.3 Sequential process behaviour

The question remaining is how to define the basic sequential behaviours of atomic processes. In previous work on DEMOS hierarchies [72,73,78,79] it was assumed that the lowest level processes were those defined by the flow of control aspects in activity diagrams (described loosely in Chapter 2). This gives a rule, in terms of that graphical formalism, that an atomic process is a *start/end* pair and all nodes connected to them by flow of control links. As a corollary, a flow of control link may not leave a process. Here, this definition is preserved, but the decomposition rules are extended, by noting that any decision to make a process atomic is arbitrary and involves an abstraction of the real system and an aggregation of underlying real world processes' behaviour. This point is especially important when trying to simplify models for more efficient solution.

3.3.1 Decomposition and Composition of Processes

A rule is introduced that atomic processes can be further decomposed in two ways: one of which merely results in finer division of their behaviour, the other in more detailed modelling. The second is dependent on the first and allows aggregation to be seen in its correct place in process based simulation. Both are independent of the particular formalism used to describe models, but here the conventions of activity diagrams are used for their convenience.

3.3.2 Breaking Down Sequential Behaviour

The introduction of this concept removes the distinction between sequential behaviour (flow of control) and synchronisation (interaction). Flow of control within a process can be represented as scheduling between two new processes. To see this consider two cases.

Case one.

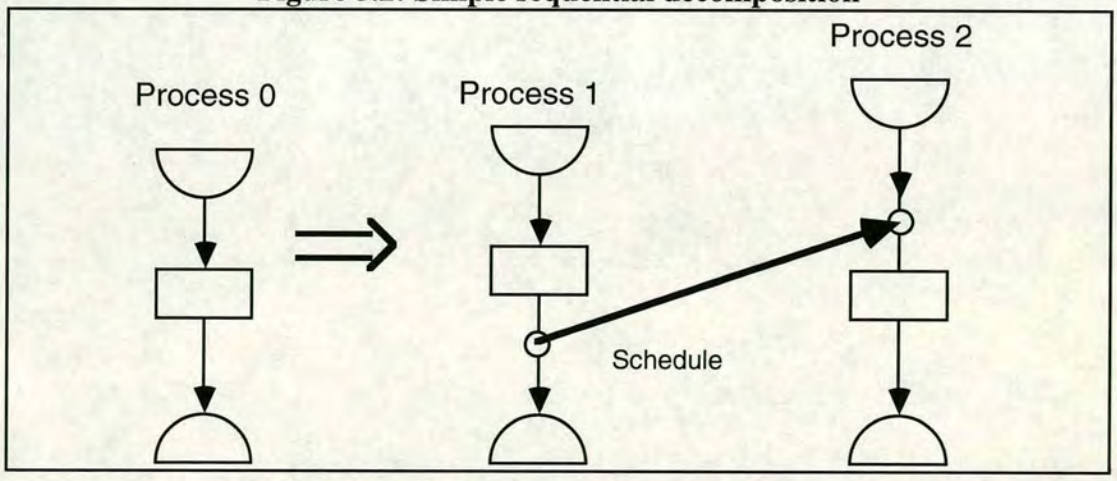
The first example in Figure 3.2 is a simple acyclic process, shown as an extended activity diagram consisting of a *start* node, a *hold* and an *end* node. This can be decomposed in general terms into two processes of the following pattern:

Process one: this has a *start* node, followed by a *hold*, followed by a *schedule* synchronisation sent to **Process two**, followed by an *end* node.

Process two: this has a *start* node, which receives the *scheduling* synchronisation from **Process one**, followed by a *hold*, followed by an *end* node.

Note that the original delay in the process being decomposed is split amongst the delay in **Process one**'s *hold*, the *scheduling* delay between **Process one** and **Process two** and the delay in the *hold* in **Process two**. In theory any of these might be zero and zero delays can be eliminated. If the times were described by stochastic delay variables, their division into component delays would require an understanding of the behaviour of the new components and of probability theory. This problem is not considered here.

Figure 3.2: Simple sequential decomposition



Case two.

The second example extends the principle to model loops in the original process. Now there is a process with a *start* node, followed by a *begin-loop* node, followed by a *hold*, followed by an *end-loop* node, followed by an *end* node. This represents a simple while loop. Its decomposition into two processes involves the substitution

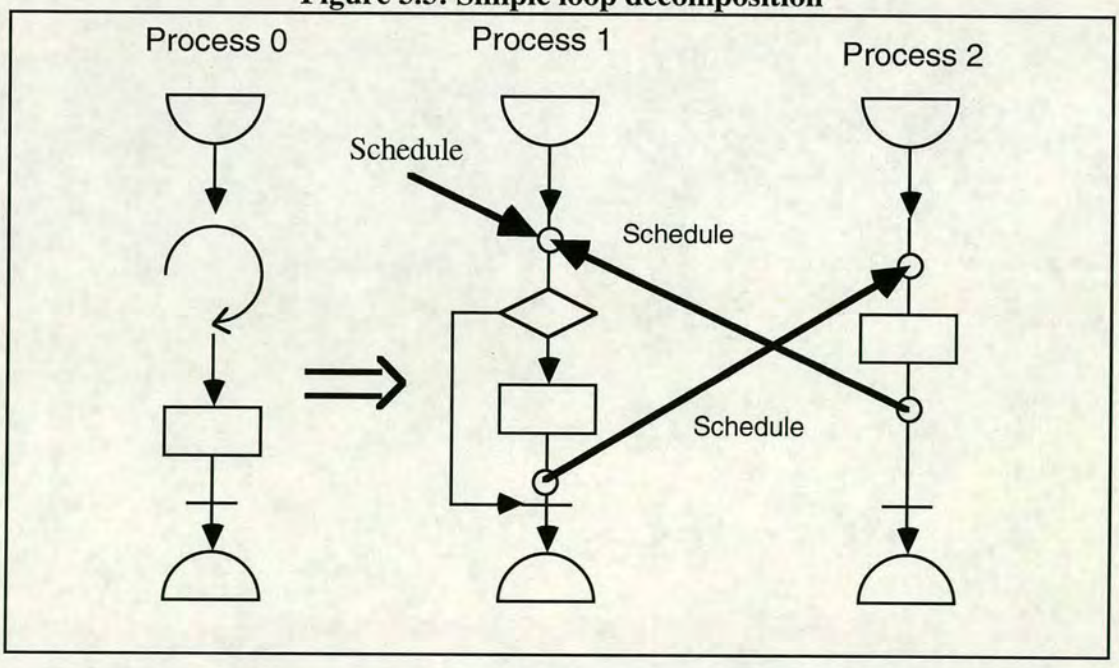
of a second process for the implied return from the *end-loop* node. This leads to the following pair of processes:

Process one: this has a *start* node, followed by an incoming *scheduling* synchronisation, followed by a *decision* node, whose outgoing links are in one case to an *end-branch* node and in the other to a *hold*, followed by a *schedule* going to **Process two**.

Process two: this has a *start* node, followed by an incoming *schedule* from **Process one**, followed by a *hold*, followed by an outgoing *schedule* going to **Process one**.

Note that here the original delay is spread across the *hold* in **Process one**, the *hold* in **Process two** and the two scheduling delays. Again this split is dependent on the way the system being described would break up the delay and any resulting zero delays can be eliminated

Figure 3.3: Simple loop decomposition



3.3.3 Delays and Aggregation

The discussion so far has concentrated on purely structural and functional decomposition of process behaviour. What would be interesting now would be to bring in the notion of aggregation of performance measures. This could lead to

integration with hierarchical experimentation and hybrid model use in experiments and to possible hybrid modelling within a single tool, in a similar manner to HIT [8,10]. These ideas are not developed further in this dissertation, but it is important to note the link for future work.

Essentially any model involves aggregation or simplification of time related behaviour. Wherever we use a simple delay, stochastic variable or formula, we choose not to model an underlying process dynamically. Thus a *hold* in DEMOS replaces a more detailed sub-model. Conversely, in a model one can replace a *hold* by a more detailed sub-component, using the sequential composition rules above. Reversing this process one can also replace a detailed part of a model with an estimate for its performance by identifying the sub-model as a process which synchronises in the appropriate way and replacing it with a hold. This is a necessary condition for aggregation.

For aggregation to be sensible and meaningful, it must also respect the condition of separability, in the sense that the processes being aggregated into a hold must interact as little as possible with the rest of the model. Ideally there should be no synchronisations between the aggregated processes and the rest of the model, apart from the scheduling ones identified as allowing reduction to a hold, i.e. there should only be a pair of schedule synchronisations, one in each direction which correspond to the start and end of the hold.

Formally this takes the work into the problems of stochastic modelling and specifically of lumpability, which are outside the scope of the present work. In practice it may be sufficient to believe that any other synchronisations are sufficiently infrequent or require so little of resources etc. that they can safely be ignored. Hillston's recent work with PEPA [37] shows that rules for aggregation can be related to equivalences in the algebraic definition of a model, which offers considerable promise for the approach developed in the rest of this dissertation.

3.4 Formal semantics for process based simulation

Having examined the concepts of process based simulation, the task is to define such ideas in a formal manner. The main concerns are to understand the behaviour of models and to reason about their properties, assuming component based modelling based on hierarchical composition. This requires the interaction among components

to be restricted to parameter passing and those mechanisms enabling synchronisation among interacting processes. Thus the first task is to find a suitable basis and the Calculus of Communicating Systems (CCS) [58] is here chosen.

The view of systems as processes is widespread in computer science. Process algebras form a major technique of concurrency theory and CCS is one of the most significant process algebras to have emerged. Previous work, for instance by Hughes and students at the University of Trondheim [Hughes, personal communication], suggested that Petri nets provided a low level formalism for flat process interaction models. Unfortunately Petri nets are not easily used in hierarchical, compositional modelling and so they are rejected here. In the end CCS was found to offer straightforward mappings for some of the mechanisms and, since better local expertise was available for it, it was chosen for further work. This was reinforced by the extension of CCS in Temporal CCS (TCCS) [98,60], which opened the possibility of including time explicitly, at least in a restricted way.

3.4.1 Modelling process interaction simulation primitives in CCS

The benefits of defining a mapping between a simulation model and a process algebra are twofold. First, it allows proper semantics to be given for the language used in simulation models and so is a step in answering the question, "How far is the simulation model actually equivalent to the system it models?" This would be especially useful if the both simulation and process algebra models could be defined using the same formalism.

Second, it is then possible to use the same notation for quantitative (performance or reliability) properties and functional (liveness, fairness etc.) properties. The desirability of this has been noted by several authors [26]. There may be limitations to the functional results that might be obtained through a given formalism, but the use of a higher level means of expressing them should encourage at least an effort in that direction. If such analysis can be mechanised, it becomes extremely attractive.

Thus, the rest of this chapter adopts a process algebra approach to the problem of defining a proper semantics for process based discrete event simulation. A common framework based on the process view of models is constructed to represent hierarchical modelling as described above. This is developed into a graphical language in Chapter 4. A tool based on such a framework allows models to be built

as a single graphical description, which can then use various combinations of simulation and functional techniques to answer both performance questions (What is the throughput under a certain load?) and functional questions (Will the system deadlock under certain assumptions?). In chapter 5 such a tool is described, building on the work in the rest of this chapter and in Chapter 4. It is used for a number of case studies in Chapter 6. In particular this tool supports discrete event simulation of such models using a language based on DEMOS [13] and functional analysis based on CCS and its timed extension (TCCS) [98], exploiting where possible the automatic reasoning support of the Concurrency Workbench [20,61].

The existing DEMOS primitives are explored initially using CCS. It is, of course, not possible to *prove* any formal equivalence between DEMOS models and CCS ones generated from the same activity diagrams, as DEMOS has no formal semantics. In fact such a semantics is being defined in expressing these equivalences. It can be argued that this is reasonable with an appeal to intuition, but it is also possible to show whether execution of DEMOS models reproduces behaviour predicted by CCS equivalent models, such as deadlocking.[1]

3.4.2 Active processes

Representations of processes map directly onto Entity declarations in DEMOS and agent definitions in CCS. By using parallel composition of agents in CCS, it is possible to instantiate co-operating and competing processes within a model in the same way as use of *new* statements in DEMOS. Interactions must be modelled in CCS by complementary actions, shared with the active or passive object involved in the interaction. In DEMOS they are calls to procedures (methods) which are attributes of those objects. In CCS internal actions are either disregarded (in un-timed models) or represented by delays matching DEMOS *hold* statements(in timed versions). Simple DEMOS sequences of actions are matched by the normal CCS prefixing of an agent with an action or a time delay. Termination, shown in DEMOS by the *end* of an Entity, is indicated in CCS by the non-temporal deadlock agent, $\underline{0}$, which performs no further actions but does not stop time passing. Figure 3.4 shows a simple example.

---

[1] Unfortunately, DEMOS itself is not entirely suited to our purposes, as we shall see, and we redefine it slightly to produce a new simulation package based more explicitly on processes for all interacting objects. The end result is a language known as *modified* DEMOS, which is described in more detail in Chapter 5.

Figure 3.4: A DEMOS sequential Entity and a corresponding TCCS agent

```
Entity class Seq;
begin
   Hammer.Acquire(1);
   Hold(3);
   Hammer.Release(1);
end;
```

$$Seq \stackrel{def}{=} \overline{hammerAcq_1}\,(3)\,\overline{hammerRel_1}\,.\underline{0}$$

Loops are represented by recursive agent definitions. Figure 3.5 shows a simple example of this.

Figure 3.5: A DEMOS repeating Entity and a corresponding TCCS agent

```
Entity class Seq;
begin
   while True do
   begin
      Hammer.Acquire(1);
      Hold(3);
      Hammer.Release(1);
   end;
end;
```

$$Seq \stackrel{def}{=} \overline{hammerAcq_1}\,(3)\,\overline{hammerRel_1}\,.Seq$$

There is a slight difficulty in defining variables. These must be modelled as agents which evolve to states where they can provide a complementary action corresponding to their current value. It is clumsy, for instance, to provide a completely general agent which performs all the actions of an integer, but it is quite straightforward to define an assignment and a value return action, which can support those functions needed in a particular case. Figure 3.6 shows a local variable in an Entity which is updated by assignment, by addition and by multiplication. Note that the definition of *Seq*, the complete entity, forbids access to the assignment action, $valAss_k$, to within *Seq*, by using the restriction operator (\). This enforces the scoping rules required for entities. Clearly the number of values, and so the number of states for *Val*, corresponds to the range of integer values and would require a huge and cumbersome expression unless the value passing version of CCS was used. For real numbers this would be worse. Thus only cases where the number of values which a

variable could take is fairly small could be handled by the Concurrency Workbench or a similar tool which generates the full state space for a model.

Figure 3.6: DEMOS Entity using a local variable and corresponding TCCS agent

```
Entity class Seq;
begin
   integer Val;
   Val := 4;
   while True do
   begin
      Val := Val + 2;
      Hold(3);
      Val := Val * 2;
   end;
end;
```

$Seq_1 \stackrel{def}{=} \overline{valAss_4}.Seq_2$

$Seq_2 \stackrel{def}{=} valGet_n.\overline{valAss_{n+2}}.Seq_3$

$Seq_3 \stackrel{def}{=} (3)valGet_m.\overline{valAss_{2\times m}}.Seq_2$

$Val_i \stackrel{def}{=} \delta.\overline{valGet_i}.Val_i + valAss_j.Val_j$

$Seq \stackrel{def}{=} (Seq_1 \mid Val_0)\backslash\{valAss_k, valGet_k : \text{MinInt} <= k <= \text{MaxInt}\}$

With a way of modelling variables, it is now easy to model conditional execution, using choices guarded by value reading actions. There are other situations in which a condition may be testable, but the principle is always the same - find out some current state value and make a choice based on it. Figure 3.7 shows a simple case involving an integer variable.

Figure 3.7: A DEMOS Entity using a local variable in a conditional choice

```
Entity class Seq;
begin integer Val;
   Val := 4;
   while True do begin
      Val := Val + 2;
      Hold(3);
      if Val<10 then Val := Val * 2 else Val := 4;
   end;
end;
```

$$Seq_1 \stackrel{\text{def}}{=} \overline{valAss_4}\,.Seq_2$$

$$Seq_2 \stackrel{\text{def}}{=} valGet_n.\overline{valAss_{n+2}}$$

$$Seq_3 \stackrel{\text{def}}{=} (3)valGet_m.\left(\sum_{m=MinInt}^{9} \overline{valAss_{2\times m}} + \sum_{m=10}^{MaxInt} \overline{valAss_4}\right).Seq_2$$

$$Val_i \stackrel{\text{def}}{=} \delta.\overline{valGet_i}\,.Val_i \quad + \quad valAss_j.Val_j$$

$$Seq \stackrel{\text{def}}{=} (Seq_1 \mid Val_0) \backslash \{valAss_k, valGet_k : \text{MinInt} <= k <= \text{MaxInt}\}$$

Conditional looping

Figure 3.8: A DEMOS Entity using a local variable in a conditional loop

```
Entity class Seq;
begin integer Val;
   Val := 4;
   while Val<10 do begin
      Val := Val + 2;
      Hold(3);
   end;
end;
```

$$Seq_1 \stackrel{\text{def}}{=} \overline{valAss_4}\,.Seq_2$$

$$Seq_2 \stackrel{\text{def}}{=} valGet_m.\left(\sum_{m=MinInt}^{9} \overline{valAss_{2+m}}\,(3)Seq_2 + \sum_{m=10}^{MaxInt} \underline{0}\right)$$

$$Val_i \stackrel{\text{def}}{=} \delta.\overline{valGet_i}\,.Val_i \quad + \quad valAss_j.Val_j$$

$$Seq \stackrel{\text{def}}{=} (Seq_1 \mid Val_0) \backslash \{valAss_k, valGet_k : \text{MinInt} <= k <= \text{MaxInt}\}$$

Conditional loops are formed as a combination of conditionals and loops, as one would expect. Figure 3.8 shows this.

Scheduling

Initial scheduling of another process is parallel composition within the scheduling agent of the remainder of its activity with an agent representing the scheduled process, prefixed in TCCS by a fixed delay. Figure 3.9 shows this.

Figure 3.9: A DEMOS Entity creating and scheduling a new Entity

```
Entity class Station;
begin
   while True do
   begin
      new Packet.Schedule(3.0);
      Hold(2.0);
   end;
end;

Entity class Packet;
begin
end;
```

Station	$\stackrel{def}{=}$	((3)*Packet* | (2)*Station*)
Packet	$\stackrel{def}{=}$	$\underline{0}$

Scheduling of a passivated process is in general modelled as a complementary action, whose receipt unblocks the passivated process. In some contexts this will form part of a larger mechanism, particularly in the context of a Wait Queue. Figure 3.10 shows the straightforward case of one process re-awakening another. Note that the scheduling is shown as a parallel composition of a terminating process consisting of the delay as a prefix and an outgoing action ($\overline{pSched}$ here) with the remaining actions of the scheduling process. This allows the delays to be interpreted correctly. Note also that receipt of a scheduling message is prefixed by δ, but sending is not, as a passivated process may wait indefinitely long before being scheduled, but a scheduling process may only act on a currently passivated process.

Figure 3.10: A DEMOS Entity scheduling a passivated Entity

```
Entity class Station;
begin
   while True do
   begin
      P1.Schedule(3.0);
      Hold(2.0);
   end;
end;

Entity class Packet;
begin
   Passivate;
end;

ref (Packet) P1;
P1:- new Packet("P1");
```

$$Station \stackrel{def}{=} ((3)\ \overline{pSched}\ .\ \underline{0} \mid (2)Station)$$

$$Packet \stackrel{def}{=} \delta.pSched\ .\ \underline{0}$$

3.4.2 Passive objects

Resources and other passive objects, which seem to correspond directly to those in DEMOS, are also modelled as agents, since CCS views all objects as active. (In Chapter 5 passive objects from DEMOS are shown re-implemented as subclasses of Entity to establish that this works.) By modelling resources as agents, blocking can be implemented for them. We now examine in turn the representation of the repertoire of process interaction synchronisation mechanisms.

Shared resource pool - Res

One obvious correspondence that holds in all the following mechanisms is that synchronisations which can block are formed by a communication, preceded by the indefinite wait (δ) in TCCS. Figure 3.11 shows this in terms of elements of the example 3.5 of Birtwistle, which was used in Chapter 2 to compare graphical formalisms.

Figure 3.11: 2 Demos Res objects used by 1 Entity and corresponding TCCS

```
entity class Ship_C;
begin
   new Ship.Schedule(4);
   ! grab 2 tugs;
   Tugs.Acquire(2);
   ! and a jetty;
   Jetties.Acquire(1);
   Hold(3);
   ! let the tugs go;
   Tugs.Release(2);
   Hold(10);
   ! ready to leave;
   Tugs.Acquire(1);
   Hold(3);
   ! clear of jetty;
   Jetties.Release(1);
   ! gone away;
   Tugs.Release(1);
end-of-Ship;

ref(Res) Jetties, Tugs;

Ship :- new Ship_c("Ship");
Tugs :- new Res("Tugs", 3);
Jetties :- new Res("Jetties", 2);
```

$$Boat \stackrel{def}{=} \delta.\overline{jAcq_1}.\delta.\overline{tugAcq_2}(T_{dk})\delta.\overline{tugRel_2}(T_{ud})\delta.\overline{tugAcq_1}(T_{dt})\delta.\overline{tugRel_1}.\delta.\overline{jRel_1}.\underline{0} \mid (T_{ArrivalsSample})Boat$$

$$Tugs_3 \stackrel{def}{=} \delta.((tugAcq_1.Tugs_2) + (tugAcq_2.Tugs_1) + (tugAcq_3.Tugs_0))$$

$$Tugs_2 \stackrel{def}{=} \delta.((tugAcq_1.Tugs_1) + (tugAcq_2.Tugs_0) + (tugRel_1.Tugs_3))$$

$$Tugs_1 \stackrel{def}{=} \delta.((tugAcq_1.Tugs_0) + (tugRel_1.Tugs_2) + (tugRel_2.Tugs_3))$$

$$Tugs_0 \stackrel{def}{=} \delta.((tugRel_1.Tugs_1) + (tugRel_2.Tugs_2) + (tugRel_3.Tugs_3))$$

$$Jetties_2 \stackrel{def}{=} \delta.((jAcq_1.Jetties_1) + (jAcq_2.Jetties_0))$$

$$Jetties_1 \stackrel{def}{=} \delta.((jAcq1.Jetties_0) + (jRel_1.Jetties_2))$$

$$Jetties_0 \stackrel{def}{=} \delta.((jRel1.Jetties_1) + (jRel_2.Jetties_2))$$

Note that in the temporal calculus it is necessary to decide whether an action is allowed to block indefinitely or to have the effect of killing the process if it cannot

be satisfied immediately. All acquire actions by processes can lead to a process being blocked, awaiting freeing of a resource and so such actions are prefixed with the indefinite waiting action δ. On the other hand, releases should only be permitted in cases where there has already been a matching acquire, leaving the matching resource always ready to accept it. Therefore releases are not prefixed with δ. Resources must be able to wait indefinitely in all states and so all their actions are prefixed with δ. Thus Figure 3.12 defines a general model of a resource in TCCS. In the basic calculus, where all actions are instantaneous, no δs are needed.

Figure 3.12: General definition of a DEMOS Res in TCCS

$$Res_0 \stackrel{def}{=} \sum_{i=1}^{Limit} \delta.resRelease_i.Res_i$$

$$Res_n \stackrel{def}{=} \sum_{i=1}^{Limit-n} \delta.resRelease_i.Res_{n+i} + \sum_{i=1}^{n} \delta.resAcquire_i.Res_{n-i}$$

$$Res_{Limit} \stackrel{def}{=} \sum_{i=1}^{Limit} \delta.resAcquire_i.Res_{Limit-i}$$

Unbounded producer/consumer - Bin

All texts on DEMOS use the Bin primitive to model producer/consumer relationships. The Bin relaxes the enforcement of a maximum amount that can be held in a shared pool and also removes the need for releases and acquires to match. An integer parameter now designates the initial amount of Widgets or whatever in the Bin when the model starts execution. This value determines the initial Bin agent to be composed in parallel with Model in the CCS version, i.e. a parameter value of n would mean using $Widgets_n$. Figure 3.13 uses an example from Birtwistle, p. 66.

Figure 3.13: Demos Bin object used by two Entitys and their corresponding TCCS agents

```
Entity class Producer;
begin
   while True do
   begin
      Hold(Make_Time);
      Wid.Give(1);
   end;
end;

Entity class Consumer;
begin
   while True do
   begin
      Wid.Take(1);
      Hold(Finish_Time);
   end;
end;
ref(Bin) Wid;
Wid :- new Bin("Widgets",0);
```

$Producer \stackrel{def}{=} (T_{Make})\ \overline{widGive_1}\ .Producer$

$Consumer \stackrel{def}{=} \overline{widTake_1}\ (T_{Finish})Consumer$

$Wid_0 \stackrel{def}{=} \delta.widGive_{MaxInt}.Wid_{MaxInt} + \delta.widGive_{MaxInt-1}.Wid_{MaxInt-1} + \ldots + \delta.widGive_1.Wid_1$

.. ..

$Wid_1 \stackrel{def}{=} \delta.widGive_{MaxInt-1}.Wid_{MaxInt} + \ldots + \delta.widTake_1.Wid_0$

$Wid_{MaxInt} \stackrel{def}{=} \delta.widTake_{MaxInt}.Wid_0 + \ldots + \delta.widTake_1.Wid_{maxInt-1}$

Figure 3.14: General form of a Bin represented in TCCS

$$Widgets_0 \stackrel{def}{=} \sum_{i=1}^{Maxint} \delta.widGive_i.Widgets_i$$

$$Widgets_n \stackrel{def}{=} \sum_{i=1}^{Maxint-n} \delta.widGive_i.Widgets_{n+i} + \sum_{i=1}^{n} \delta.widTake_i.Widgets_{n-i}$$

$$Widgets_{Maxint} \stackrel{def}{=} \sum_{i=1}^{Maxint} \delta.widTake_i.Widgets_{Maxint-i}$$

As a bin is unbounded, there is a different problem to that for representing a resource. The general form of the Widget bin would have to be given as a set of agents, one for each value from 1 to the practical upper limit to the capacity of a Bin, here written as MaxInt. In theory it should be infinity. Whatever the effective upper limit of a Bin, there is an extremely large state space to represent. What is more, in every current level of occupation n, i is free to range over $1..n$. It is, therefore, necessary to use the value passing calculus and great difficulties arise if it is desirable to resort to the Concurrency Workbench.

In most cases it will actually be possible to limit the capacity of the Bin, making it into a bounded buffer, as described below. In nearly all cases, it will be possible to limit the set of possible values for i, at least removing transitions and, often, states. These possibilities are discussed in Chapter 6. It is probably unwise to use the DEMOS Bin, except when unavoidable.

Bounded buffer - Store

As mentioned above, the unbounded Bin is problematical as a modelling device in simulation. It is generally better to use a bounded buffer. In practice this is usually more accurate, anyway, as all physical systems have limited buffer space and it is often the purpose of simulation modelling to optimise the use of such buffering. Birtwistle's DEMOS does not have finite capacity buffers, but they are added to *modified* DEMOS, which is fully described in Chapter 5, where they are known as class Store. Using this construct, the producer/consumer interaction can be re-modelled as shown in Figure 3.15.

Figure 3.15: Demos Store object used by two Entitys and their corresponding TCCS agents

```
Entity class Producer;
begin
   Hold(Make_Time);
   Widgets.Add(1);
   repeat;
end;

Entity class Consumer;
begin
   Widgets.Remove(1);
   Hold(Finish_Time);
   repeat;
end;

ref(Store) Widgets;

Widgets :- new Store("Widgets",4,0);
```

$$Prod \stackrel{def}{=} (T_{Make})\,\overline{widAdd_1}\,.Producer$$

$$Cons \stackrel{def}{=} widRem_1(T_{Finish})Consumer$$

$$Wid_4 \stackrel{def}{=} \delta.widRem_4.Wid_0 + \delta.widRem_3.Wid_1 + \delta.widRem_2.Wid_2 + \delta.widRem_1.Wid_3$$

$$Wid_3 \stackrel{def}{=} \delta.widAdd_1.Wid_4 + \delta.widRem_3.Wid_0 + \delta.widRem_2.Wid_1 + \delta.widRem_1.Wid_2$$

$$Wid_2 \stackrel{def}{=} \delta.widAdd_2.Wid_4 + \delta.widAdd_1.Wid_3 + \delta.widRem_2.Wid_0 + \delta.widRem_1.Wid_1$$

$$Wid_1 \stackrel{def}{=} \delta.widAdd_3.Wid_4 + \delta.widAdd_2.Wid_3 + \delta.widAdd_1.Wid_2 + \delta.widRem_1.Wid_0$$

$$Wid_0 \stackrel{def}{=} \delta.widAdd_4.Wid_4 + \delta.widAdd_3.Wid_3 + \delta.widAdd_2.Wid_2 + \delta.widAdd_1.Wid_1$$

As a Store is bounded, it is a similar problem to representing a resource. The general form of a Widget Store is a finite summation of choices, shown in Figure 3.16. This can be simplified in many models, including our example, as shown in Chapter 6. *Limit* is the physical upper limit to the capacity of a Store. It is the value of the second parameter of the Store instantiation. The third parameter is the initial number of items in the Store.

Figure 3.16: General form of a Store object represented in TCCS

$$Widgets_0 \stackrel{def}{=} \sum_{i=1}^{Limit} \delta.widAdd_i.Widgets_i$$

$$Widgets_n \stackrel{def}{=} \sum_{i=1}^{Limit-n} \delta.widAdd_i.Widgets_{n+i} + \sum_{i=1}^{n} \delta.widRem_i.Widgets_{n-i}$$

$$Widgets_{Limit} \stackrel{def}{=} \sum_{i=1}^{Limit} \delta.widRem_i.Widgets_{Limit-i}$$

First In First Out (FIFO) Queue

A number of explicit queueing mechanisms are defined for a DEMOS Entity. In *modified* DEMOS another queue, for passive objects holding values, known as Messages, is added. In DEMOS all Entitys are removed from queues in the order of highest priority. In the time ordered event list, the next event time acts as the reciprocal of the priority. Those with the same priority are removed in the same order that they were added. This is in effect an implementation of multiple FIFO queues, with higher priority queues polled first. In *modified* DEMOS, Messages are also removed in FIFO order. The importance of such an implementation of waiting is that reproducibility is guaranteed. In the class hierarchy of DEMOS it is possible to define a parent class for all queues which implements a FIFO discipline.[1] Thus CCS must be able to represent a FIFO queue mechanism.

Milner [Milner 1990] gives the following specification for a FIFO queue (Chapter 6, p135):

$$Queue(\varepsilon) \stackrel{def}{=} in(x).Queue(x) + empty.Queue(\varepsilon)$$

$$Queue(s{:}v) \stackrel{def}{=} in(x).Queue(x{:}s{:}v) + \overline{out}\,(v).Queue(s)$$

Defining Milner's linking operator, $\frown$, by:

$$P \frown Q = (P[i'/i, e'/e, o'/o] \mid Q[i'/out, e'/empty, o'/in]) \setminus \{i', e', o'\}$$

a FIFO queue can be implemented as:

---

[1]This is actually a fiction. For various implementation reasons, DEMOS implements some queues independently of the inheritance structure.

$$FIFO\langle v_1,\ldots,v_n\rangle \stackrel{def}{=} C(v_n)^\frown\ldots^\frown C(v_1)^\frown B$$

where v_1 is the last item to enter the queue and v_n is the first and

$$B \stackrel{def}{=} in(x).(C(x)^\frown B) + empty.B$$

$$C(x) \stackrel{def}{=} in(y).\overline{o}\,(y).C(x) + \overline{out}\,(x).D$$

$$D \stackrel{def}{=} \overline{e}\,.B + i(x).C(x)$$

This uses the value passing calculus and so allows x and y to have an infinite range of values. In a simulation model, the potential values of x and y would be constrained to the set of identifying tags, one of which would be associated uniquely with each process in a model. This might in many cases be provably finite *a priori*, but could not be guaranteed to be so for all models.

Master/slave - WaitQ/Coopt

The most general mechanisms in DEMOS are the WaitUntil and the master/slave *coopt/schedule* mechanisms. Here the master/slave mechanism is considered. This requires a double queue in DEMOS, one for slave processes, which become passive and *wait* in a queue until *coopted* and re-*scheduled* by a master process, and one for master processes, which wait implicitly until they can *coopt* a slave and may then re-*schedule* it whenever they are finished with it.

The example shown in Figure 3.17 is a simple ferry model, where cars are the slaves and ferries the masters. Cars are independent until they reach the harbour, when they wait in a ferry queue until a ferry coopts them and eventually re-schedules them to continue after their voyage. Ferries are always independent, loading (coopting) cars and transporting them to their destination, and unloading (scheduling) them .

Figure 3.17: Master and slave Entitys with a WaitQ and their CCS representation

```
Entity class Car;
begin
   new Car("Car").Schedule(ArrivalTime);
   Hold(TripTime1);
   FerryQueue.Wait;
end;

Entity class Ferry;
begin
   ref(Car) Cargo;
   while True do
   begin
      Cargo :- FerryQueue.Coopt;
      Hold(VoyageTime1);
      Cargo.Schedule(0);
      Hold(VoyageTime2);
   end;
end;

ref(WaitQ) FerryQueue;

FerryQueue:- new WaitQ("Ferries");
```

$Ferry$	$\stackrel{def}{=}$	$\sum_{i=1}^{MaxInt} cooptFQ_n(T_{V1})\,\overline{sched_n}\,(T_{V2})Ferry$
Car_n	$\stackrel{def}{=}$	$(T_{Tr1})\,\overline{waitFQ_n}\,.sched_n.0 \mid (T_{Arr})carGet_k.Car_k$
$CarNo_i$	$\stackrel{def}{=}$	$\overline{carGet_i}\,.CarNo_{i+1}$
$FQ<>$	$\stackrel{def}{=}$	$waitFQ_n.FQ<n>$
$FQ<n,L>$ list of integers	$\stackrel{def}{=}$	$waitFQ_k.FQ<n,L,k> + \overline{cooptFQ_n}\,.FQ<L>$ L is any

The WaitQ is the first explicit use of a queue in any of the mechanisms modelled. It is shown using a convenient shorthand form of CCS, where agents are subscripted with ordered lists of integers. This allows queueing disciplines, such as the First Come First Served (FCFS) (more often known as First In First Out (FIFO)) one assumed for the ferry, to be represented concisely. The underlying implementation of a FIFO queue was presented above.

The model shown also defines a unique numbering for each car and shows it being generated explicitly. This corresponds to the ability within DEMOS to locate each instance of an Entity class through a reference variable, which holds its location within the SIMULA heap. For convenience, this numbering will sometimes be assumed without being generated explicitly.

Figure 3.18 general CCS representation of a WaitQ

$Wq<> \stackrel{def}{=} wait_n.Wq<n> + empty.Wq<>$

$Wq<n,L> \stackrel{def}{=} wait_k.Wq<n,L,k> + \overline{coopt_n}.Wq<L>$ L is any list of integers

Signalling changes in conditions - CondQ/Signal

DEMOS implements the concept of a conditional wait, which can be thought of as a generalisation of waiting for a resource. An Entity can perform a WaitUntil, which is a procedure requiring a particular condition to be true. This will block the Entity in a nominated CondQ until that condition holds. Some simulation packages, such as SIMON [33], use the general notion of WaitUntil for all blocking and synchronisation. This general mechanism requires that all conditions be re-tested by a central monitor process every time a state change occurs. This is extremely inefficient, as only those conditions affected by the change need be re-tested. DEMOS instead requires that an Entity which causes a state change relevant to a blocked Entity in a CondQ, performs a Signal on that queue. This makes it the responsibility of the modeller to ensure that all state changes are understood in relation to any conditional waiting and to insert appropriate Signal calls.

The wait for a condition can be easily implemented as half of a complementary action, which will be matched by some agent when the condition is satisfied. This is similar to the implementation of an if condition in section 3.3.2, but does not involve a choice. Figure 3.19 shows a simple example of such waiting.

Figure 3.19: An Entity waiting on a condition and an Entity signalling a change through a CondQ

```
Entity class Waiter;
begin
   CQ.WaitUntil(Val=3);
end;

Entity class Signaller;
begin
   while True do
   begin
      Val := Val + 1;
      CQ.Signal;
   end;
end;

integer Val;

ref(CondQ) CQ;
CQ :- new CondQ("CQ");
```

$$Waiter \stackrel{def}{=} \delta.waitCQ.(valGet_3.\underline{0} + \sum_{i=MinInt}^{2} valGet_i\,.Waiter + \sum_{i=4}^{MaxInt} valGet_i\,.Waiter)$$

$$Val_i \stackrel{def}{=} \overline{valGet_i}\,.Val_i + valAss_j.Val_j$$

$$Signaller \stackrel{def}{=} valGet_n.\overline{valAss_{n+1}}\,.\,\overline{waitCQ}\,.Signaller$$

$$Model \stackrel{def}{=} (Waiter \mid Signaller \mid Val_0\,) \setminus \{valGet_{MinInt}, .., valGet_{MaxInt}, valAss_{MinInt}, .. , valAss_{MaxInt}\}$$

This naïve implementation has certain limitations. In particular it only allows one Entity to proceed when a state change occurs. The DEMOS CondQ has two modes of operation, controlled by a Boolean called All. If All is set to False (the default), triggering of Entitys continues after a Signal until the first one in the CondQ fails its condition. If All is set to True, triggering always continues to the end of the list. All those which pass the test are scheduled immediately after the signalling Entity. Those which fail return to the same place in the CondQ.

3.20: The version of the simple model with All set to False

$$Waiter_n \stackrel{def}{=} valGet_3.0 + \sum_{i=MinInt}^{2} \overline{waitCQ_n}\ .Waiting_n + \sum_{i=4}^{MaxInt} \overline{waitCQ_n}\ .Waiting_n$$

$$Waiting_n \stackrel{def}{=} try_n.\left(valGet_3.\ \overline{goGo_n}\ .0 + \sum_{i=MinInt}^{2} valGet_i.Failed_n + \sum_{i=4}^{MaxInt} valGet_i.Failed_n \right)$$

$$Failed_n \stackrel{def}{=} \overline{noGo_n}\ .Waiting_n$$

$$Val_i \stackrel{def}{=} \overline{valGet_i}\ .Val_i + valAss_j.Val_j$$

$$Signaller \stackrel{def}{=} valGet_n.\ \overline{valAss_{n+1}}\ .\ \overline{signalCQ}\ .done.Signaller$$

$$CQ\langle \varepsilon \rangle \stackrel{def}{=} \overline{empty}\ .CQ\langle \varepsilon \rangle + signalCQ.CQ\langle \varepsilon \rangle + waitCQ_n.CQ\langle n \rangle$$

$$CQ\langle s{:}V \rangle \stackrel{def}{=} signalCQ.Try\langle s{:}V, \varepsilon \rangle + waitCQ_n.CQ\langle s{:}V{:}n \rangle$$

$$Try\langle s{:}V, W \rangle \stackrel{def}{=} \overline{try_s}\ .\left(noGo_s.\ \overline{done}\ .CQ\langle W{:}\ s\ {:}V \rangle + goGo_s.Try\langle V, W \rangle\right)$$

$$Try\langle \varepsilon, W \rangle \stackrel{def}{=} \overline{done}\ .CQ\langle W \rangle$$

Angle brackets denote lists of lists in which lower case letters are singletons, upper case letters are lists, ":" is concatenation and ε is the empty list.

Figure 3.21: The version of the simple model with All set to true

$$Waiter_n \stackrel{def}{=} valGet_3.0 + \sum_{i=MinInt}^{2} \overline{waitCQ_n}.Waiting_n + \sum_{i=4}^{MaxInt} \overline{waitCQ_n}.Waiting_n$$

$$Waiting_n \stackrel{def}{=} try_n.\left(valGet_3.\overline{goGo_n}.0 + \sum_{i=MinInt}^{2} valGet_i.Failed_n + \sum_{i=4}^{MaxInt} valGet_i.Failed_n \right)$$

$$Failed_n \stackrel{def}{=} \overline{noGo_n}.Waiting_n$$

$$Val_i \stackrel{def}{=} \overline{valGet_i}.Val_i + valAss_j.Val_j$$

$$Signaller \stackrel{def}{=} valGet_n.\overline{valAss_{n+1}}.\overline{signalCQ}.done.Signaller$$

$$CQ\langle\varepsilon\rangle \stackrel{def}{=} \overline{empty}.CQ\langle\varepsilon\rangle + signalCQ.CQ\langle\varepsilon\rangle + waitCQ_n.CQ\langle n\rangle$$

$$CQ\langle s{:}V\rangle \stackrel{def}{=} signalCQ.Try\langle s{:}V, \varepsilon\rangle + waitCQ_n.CQ\langle s{:}V{:}n\rangle$$

$$Try\langle s{:}V, W\rangle \stackrel{def}{=} \overline{try_s}.\left(noGo_s.Try\langle V, W{:}s\rangle + goGo_s.Try\langle V, W\rangle\right)$$

$$Try\langle\varepsilon, W\rangle \stackrel{def}{=} \overline{done}.CQ\langle W\rangle$$

Angle brackets denote lists of lists in which lower case letters are singletons, upper case letters are lists, ":" is concatenation and ε is the empty list.

This looks quite complicated and its implementation in the basic calculus would be long winded, but it can be built relatively simply from a pair of FIFO queues, corresponding to the two lists which parameterise *Try*.

Interrupt

DEMOS allows one Entity to break into a hold in another. Once interrupted by a call of Interrupt with an integer parameter, the interrupted Entity can choose how to proceed based on this value. This mechanism is not straightforward to represent in CCS, as it relies on one Entity remaining in an interruptable state for an interval of time and, having reached the end of this, proceeding. Figure 3.22 shows a simple example in DEMOS and CCS, using a small grain of time (eps) between each check for the interrupt. This could be argued to be what a simulation effectively does, since reals are held as discrete values in a digital computer, but is essentially a costly and coarse approximation.

Figure 3.22: One Entity interrupting another

```
Entity class Interrupted;
begin
   Hold(TDo);
   if Interrupt=3 then new
Interrupted("Ited").Schedule(0);
end;

Entity class Interrupter;
begin
   Ited.Interrupt(3);
end;

Ited :- new Interrupted("Ited");
Iter :- new Interrupter("Iter");
```

$$Ited \stackrel{def}{=} Checker_{3 \div eps}$$

$$Checker_t \stackrel{def}{=} (eps)\left(iGet_3.Ited + \left(\sum_{i=MinInt}^{2} iGet_i + \sum_{i=4}^{MaxInt} iGet_i \right).Checker_{t-1} \right) \quad t > 0$$

$$Checker_0 \stackrel{def}{=} 0$$

$$Iter \stackrel{def}{=} (eps)\,\overline{iGet_0}\,(eps)...(eps)\,\overline{iGet_3}\,.0$$

Message queues

For modelling convenience and efficiency of model execution, *modified* DEMOS includes a FIFO queue of passive objects which carry information. This presents no problems for CCS, as the FIFO and the local attribute have both been dealt with above. To save space, the message queue is not shown in any detail here.

3.4.4 Building complete models

There remains the question of how to represent models and sub-models within this formal framework. This turns out to be very straightforward.

Overall model definition

In general a model in CCS can be defined as the parallel composition of the model environment (ENV) with the agents making up the model (MODEL). ENV will behave differently depending on the type of execution chosen, e.g. replications or single run. Here it is treated as a simple passage of time.

The complete CCS model must also restrict the visibility of those actions which are fully defined by the processes and resources present. This means all actions for a complete model. Such restriction corresponds to the notion of satisfaction of visible links when matching synchronisations in section 3.2. Figure 3.23 shows a model built of the Boat, Jetties and Tugs agents from the harbour model. The convention is adopted, used throughout this dissertation, that restriction of all remaining visible labels be denoted by \L(*MODEL*)

Figure 3.23: Defining a complete model in CCS

DEMOS	$\stackrel{def}{=}$	$(T_{sim})\ .\ 0$
MODEL	$\stackrel{def}{=}$	$TUGS_3 \mid JETTIES_2 \mid BOAT$
PROG	$\stackrel{def}{=}$	$(DEMOS \mid MODEL) \backslash L(MODEL)$

Building hierarchies

The use of hierarchically defined sub-models in DEMOS corresponds to parallel composition and label restriction in CCS. The principal difference between their use now to define sub-model processes and above to define a complete model is that only those labels which correspond to actions contained within the component process are now restricted. These hidden actions become either τs or, as described in Chapter 6, can be eliminated by applying the expansion law. This is the equivalent of the graphical convention of drawing a box round the hidden parts of the compound process in the graphical conventions of Extended Activity Diagrams.

The question of satisfied but accessible actions, where the compound process provides matches for synchronisations which are still open to outside processes, is simply resolved. Their labels are not hidden. Note, however, that CCS does not allow us to define the maximum or minimum arity of such communication groups. It only deals in the possibility or prohibition of engaging in actions on a one to one basis.

In a corresponding DEMOS source program, visible CCS labels correspond to DEMOS parameters propagating out to higher textual levels. When binding particular instances of agents together, re-labelling is used, creating matching private names for those actions which provide the linking. To show these features, consider

the Dining Philosophers model as a simple example. This model consists of identical Philosopher processes linked in a ring by shared Fork resources. This means in DEMOS that two resources are parameters (type ref(Res)) of each Entity, which bind the Entity to instantiated resource objects in the complete model.

3.24: The hierarchical model of the Dining Philosophers

```
EXTERNAL class DEMOS;
DEMOS class E_DEMOS;
begin

   Entity class Philosopher(Right_Fork, Left_Fork,T_Feed, T_Think);
              ref(Res) Right_Fork, Left_Fork; REAL T_Feed, T_Think;
   begin
      while True do
      begin
         Right_Fork.acquire(1);
         Hold(0.2);
         Left_Fork.acquire(1);
         Hold(T_Feed);
         Right_Fork.release(1);
         Left_Fork.release(1);
         Hold(T_Think);
      end;
   end of Philosopher;

end of E_DEMOS;

begin
   EXTERNAL class E_DEMOS;
   E_DEMOS
   begin
      ref(Res) Fork1, Fork2, Fork3;
      real I_T_Feed, I_T_Think;

      I_T_Feed := InReal; I_T_Think := InReal;

      Fork1 :- new Res("Fork",1);
      Fork2 :- new Res("Fork",1);
      Fork3 :- new Res("Fork",1);
      new Philosopher("P",Fork1, Fork2, I_T_Feed,I_T_Think).Schedule(0.0);
      new Philosopher("P",Fork2, Fork3, I_T_Feed,I_T_Think).Schedule(0.0);
      new Philosopher("P",Fork3, Fork1, I_T_Feed,I_T_Think).Schedule(0.0);
      Hold(100.0);
   end;
end
```

The CCS model restricts for the Philosopher agents anything except the communication actions with the Fork resources, which is the same as making the Fork a parameter. It then uses re-labelling to bind the Philosopher agents to the correct Fork agents. Finally all these labels are restricted in the complete model. Since the forks are shared by philosophers their actions are not restricted, merely used in renaming. Restriction would make the forks private to a philosopher.

Figure 3.25: CCS hierarchical model of the dining philosophers

$$Philosopher \stackrel{def}{=} \overline{rfAcq_1}\,(2)\,\overline{lfAcq_1}\,(T_{feed})\,\overline{rfRel_1}\;\overline{lfRel_1}\,(T_{think})Philosopher$$

In fact there are no synchronisations to be hidden at this level.

$$Fork \stackrel{def}{=} fAcq_1.NoFork$$

$$NoFork \stackrel{def}{=} fRel_1.Fork$$

$$P_1 \stackrel{def}{=} Philosopher[a1/rfAcq_1, a2/lfAcq_1, r1/rfRel_1, r2/lfRel_1]$$

$$P_2 \stackrel{def}{=} Philosopher[a2/rfAcq_1, a3/lfAcq_1, r2/rfRel_1, r3/lfRel_1]$$

$$P_3 \stackrel{def}{=} Philosopher[a3/rfAcq_1, a1/lfAcq_1, r3/rfRel_1, r1/lfRel_1]$$

$$Fork_1 \stackrel{def}{=} Fork[a1/fAcq_1, r1/fRel_1]$$

$$Fork_2 \stackrel{def}{=} Fork[a2/fAcq_1, r2/fRel_1]$$

$$Fork_3 \stackrel{def}{=} Fork[a3/fAcq_1, r3/fRel_1]$$

$$Model \stackrel{def}{=} (Fork_1 \mid Fork_2 \mid Fork_3 \mid P_1 \mid P_2 \mid P_3) \setminus \{a1,a2,a3,r1,r2,r3\}$$

3.5 Validating the CCS definition of DEMOS primitives

The definitions in section 3.4 have presented a CCS description of all the mechanisms present in DEMOS and added some new ones which seem useful and which will be implemented in the graphical formalism and packages described in Chapters 4 and 5. The formalisation of the semantics of process based models, including hierarchical models, is thus apparently complete. There remains the important question of whether the actual behaviour of DEMOS matches that predicted by the CCS definitions. If it does, the semantics given can be applied to reasoning about existing DEMOS models. If not, the extent of its applicability must be defined and the possibility of re-implementing some parts of DEMOS, in addition to the extensions already made, must be considered.

The approach taken to validate the definitions given in section 3.4 was to consider a number of representative models expressed in both ways and to compare their behaviour. The CCS model was used to predict the required behaviour of the

DEMOS version. In most cases the definitions given proved accurate. Evidence from these is given in Appendices B (DEMOS models) and C (CCS models). Only the principal anomaly is discussed here, but it exemplifies the general approach.

3.5.1 Validating resource contention for DEMOS

To investigate mapping of anonymous resource contention, as defined in section 3.4.2, into both DEMOS and CCS, the harbour model given in [13] is used. For this example the DEMOS source code and the Concurrency Workbench compatible CCS were generated by the graphical modelling tool described in Chapter 5. For this the whole of PROG, as in section 3.4.4, is developed. It reverses the initial acquires of jetties and tugs by the boat compared to the version in Birtwistle. This is claimed to deadlock, while the original does not. This should be shown by the TCCS version.

Concurrency Workbench Model

To evaluate the initial representation in the two languages, the encoded TCCS model, with the supposed deadlock potential, was fed into the Edinburgh Concurrency Workbench and a trace of the simulation of that model produced. The source and trace are given below. The sequence of actions in the CWB was not the same as in DEMOS . Importantly, the DEMOS model did deadlock after ship 3 had seized two tugs at 8.00, while the CCS model still allowed ships to dock and depart.

Figure 3.26: Demos source code for the "deadlocking" harbour model

```
begin external class demos; DEMOS begin
      entity class Ship_C;
      begin new Ship.Schedule(4);
         ! grab 2 tugs; Tugs.Acquire(2);
         ! and a jetty; Jetties.Acquire(1);
         Hold(3);
         ! let the tugs go; Tugs.Release(2);
         Hold(10);
         ! ready to leave; Tugs.Acquire(1);
         Hold(3);
         ! clear of jetty; Jetties.Release(1);
         ! gone away; Tugs.Release(1);
      end-of-Ship;
      ref(Ship_C) Ship; ref(Res) Jetties; ref(Res) Tugs;
      Ship :- new Ship_c("Ship");
      Tugs :- new Res("Tugs", 3);
      Jetties :- new Res("Jetties", 2);
      Ship.Schedule(0.0); Hold(100);
   end
end
```

Figure 3.27: "Deadlocking" harbour modelled in TCCS.

BOAT	$\stackrel{def}{=}$	$\delta.\overline{jAcq_1}.\delta.\overline{tugAcq_2}\ (3)\ \overline{tugRel_2}\ (10)\delta.\overline{tugAcq_1}\ (3)\ \overline{tugRel_1}.\overline{jRel_1}$ $\mid (4)BOAT$
$TUGS_3$	$\stackrel{def}{=}$	$\delta.((tugAcq_1.TUGS_2) + (tugAcq_2.TUGS_1) + (tugAcq_3.TUGS_0))$
$TUGS_2$	$\stackrel{def}{=}$	$\delta.((tugAcq_1.TUGS_1) + (tugAcq_2.TUGS_0) + (tugRel_1.TUGS_3))$
$TUGS_1$	$\stackrel{def}{=}$	$\delta.((tugAcq_1.TUGS_0) + (tugRel_1.TUGS_2) + (tugRel_2.TUGS_3))$
$TUGS_0$	$\stackrel{def}{=}$	$\delta.((tugRel_1.TUGS_1) + (tugRel_2.TUGS_2) + (tugRel_3.TUGS_3))$
$JETTIES_2$	$\stackrel{def}{=}$	$\delta.((jAcq_1.JETTIES_1) + (jAcq_2.JETTIES_0))$
$JETTIES_1$	$\stackrel{def}{=}$	$\delta.((jAcq_1.JETTIES_0) + (jRel_1.JETTIES_2))$
$JETTIES_0$	$\stackrel{def}{=}$	$\delta.((jRel_1.JETTIES_1) + (jRel_2.JETTIES_2))$

Note that the agent \$0 (non-temporal deadlock) is introduced to prevent premature deadlock by allowing terminated processes to idle, in CCS terms. Note also that the Obs agent and the action *n* are introduced to allow us to observe ships being created in the CCS trace.

Figure 3.28: Concurrency workbench model of harbour

```
bi Boat $'tugacq2.$'jacq1.(Work|NewBoat)
bi Work 3.'tugrel2.10.$'tugacq1.3.'tugrel1.'jrel1.$0
bi NewBoat 4.'n.Boat

bi Tugs3 ($tugacq1.Tugs2)+($tugacq2.Tugs1)+($tugacq3.Tugs0)
bi Tugs2 ($tugacq1.Tugs1)+($tugacq2.Tugs0)+($tugrel1.Tugs3)
bi Tugs1 ($tugacq1.Tugs0)+($tugrel1.Tugs2)+($tugrel2.Tugs3)
bi Tugs0 ($tugrel1.Tugs1)+($tugrel2.Tugs2)+($tugrel3.Tugs3)

bi Jetty2 ($jacq1.Jetty1) + ($jacq2.Jetty0)
bi Jetty1 ($jacq1.Jetty0) + ($jrel1.Jetty2)
bi Jetty0 ($jrel1.Jetty1) + ($jrel2.Jetty2)

bi Obs $n.Obs
bi DEMOS Obs|100.0
bi Model (Tugs3 | Jetty2 | Boat)
\{tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3,\
jacq1,jacq2,jrel1,jrel2}
bi Prog (DEMOS | Model)\{n}
```

Figure 3.29: Traces from CWB and DEMOS for harbour model

Output from Concurrency Workbench

```
The Edinburgh Concurrency Workbench
(Version 6.12, April 15, 1993)

Sim>   --- t<tugacq2> --->
  --- 1 --->
  --- t<jacq1> --->
  --- 1 --->
  --- 1 --->
  --- 1 --->
  --- t<tugrel2> --->
  --- 1 --->
  --- t<n> --->
  --- t<tugacq2> --->
  --- t<jacq1> --->
  --- 1 --->
  --- 1 --->
  --- 1 --->
  --- t<tugrel2> --->
  --- 1 --->
  --- t<n> --->
  --- t<tugacq2> --->
  --- 1 --->
  --- 1 --->
  --- 1 --->
  --- 1 --->
  --- 1 --->
  --- t<tugacq1> --->
```

Trace using DEMOS

```
TIME/ CURRENT AND ITS ACTION(S)

0.00 DEMOS HOLDS FOR 100.00, UNTIL 100.0
  Ship 1   SCHEDULES Ship 2 AT 4.000
       SEIZES 2 OF Tugs
       SEIZES 1 OF Jetties
       HOLDS FOR 3.000, UNTIL 3.000
3.000      RELEASES 2 TO Tugs
       HOLDS FOR 10.000, UNTIL 13.000
4.000 Ship 2   SCHEDULES Ship 3 AT 8.000
       SEIZES 2 OF Tugs
       SEIZES 1 OF Jetties
       HOLDS FOR 3.000, UNTIL 7.000
7.000      RELEASES 2 TO Tugs
       HOLDS FOR 10.000, UNTIL 17.000
8.000 Ship 3  SCHEDULES Ship 4 AT 12.000
       SEIZES 2 OF Tugs
       AWAITS 1 OF Jetties
12.000 Ship 4 SCHEDULES Ship 5 AT 16.000
       AWAITS 2 OF Tugs
```

The traces are clearly different. This brings into question the whole approach proposed. The question is whether the behaviour of the system has been incorrectly modelled when generating the CCS model or whether DEMOS fails to implement the required semantics.

The CCS model shows two major differences. First, it allows a unit of time to pass between the tugacq2 and jacq1 actions at the start of the CWB trace. This shows that time passing is treated as an action of equal priority with "real" actions prefixed with δs. Although this does not affect the outcome of the present model, it must be carefully monitored in cases where time is explicitly used. Second, the DEMOS model becomes deadlocked, as claimed, after time 8.00, when ship 3 has seized two tugs. The CCS model instead allows a further tugacq1, which release the jetty and removes the potential for deadlock.

These problems concern the non-determinism of CCS choices. Thus, although agents may have been generated in a certain sequence during the evolution of a model, there

is nothing in CCS to guarantee which ones will act first. This will be a problem wherever an event triggers a state change which could non-deterministically enable several pending processes. In fact even the non-deadlocking version of this model shows different behaviour in the two descriptions.

What must really be done in CCS to model the behaviour of the DEMOS model? On close examination of its definition, DEMOS implements Acquire as operating on a first come first served basis, even if some processes, requiring smaller amounts of a resource but arriving later, are thereby blocked unnecessarily. This alters the sequence of events in the model significantly. To support the DEMOS view of Acquire, any such action might be seen as taking as much resource as is available at the time and waiting for more to become available. This is not the same as making CCS acquires into a sequence of unit acquires, unfortunately, since this does not block other processes from subsequently jumping the queue when a release occurs. Thus, a FIFO queueing mechanism, described below, would be needed.

The alternative is to redefine Acquire in the discrete event simulation package as operating in line with the CCS semantics of section 3.4.2 and leave a greedy option for the cases where the current implementation is useful. This seems more likely to avoid confusion. Indeed in most cases it appears unreasonable to prevent processes requiring smaller amounts of a resource from proceeding, unless there is an explicit resource management mechanism designed to achieve this in the actual system being modelled. From the Chapter 4 onwards with a rewritten version including both options, with non-queueing as the default, is assumed not just for resources, but for Bins and Stores. This forms part of *modified* DEMOS as presented in chapter 5.

First In First Out (FIFO) Resource

To model the actual behaviour of a DEMOS Res requires a much more complex version of resources, using the FIFO mechanism described in section 3.4.2 to control access. The importance of such an implementation of waiting is that greater reproducibility is guaranteed in the CCS model, as was noted for WaitQs. It also has the side effect of enforcing fairness in models, both DEMOS and CCS, which might otherwise produce starvation.

Modelling the harbour system using a FIFO queue for resources can reproduce the behaviour of the original DEMOS model, except for the non-deterministic passing of

time. The principal difference is that all Acquire requests are now tagged with a unique identifier for the process which is trying to acquire the resource. This is inserted into the resource's FIFO queue and the resource only carries out its side of Acquire requests tagged with that identifier. The harbour model now looks as shown in Figure 3.30.

Figure 3.30: FIFO resource version of harbour model in CCS

$$Boat_n \stackrel{def}{=} \overline{jAcq_{n,1}}\ .jGot_{n,1}.\ \overline{tugAcq_{n,2}}\ .tGot_{n,2}.\ \overline{tugRel_{n,2}}\ .\ \overline{tugAcq_{n,1}}\ .tGot_{n,1}.\ \overline{tugRel_{n,1}}\ .\ \overline{jRel_{n,1}}$$

$$Tugs<>_{NTugs} \stackrel{def}{=} \sum_{n=1}^{NTugs} tAcq_{j,n}.\ \overline{tGot_{j,n}}\ .Tugs<>_{NTugs-n})$$

$$Tugs<>_k \stackrel{def}{=}$$

$$\sum_{n=1}^{k} tAcq_{j,n}.\ \overline{tGot_{j,n}}\ .Tugs<>_{k-n} \quad + \quad \sum_{n=k+1}^{NTugs} tAcq_{j,n}.Tugs<[j,n]>_k$$

$$+ \quad \sum_{m=1}^{NTugs-k} tugRel_m.Tugs<>_{(k+m)}$$

$$0<k<NTugs$$

$$Tugs<>_0 \stackrel{def}{=} \sum_{m=1}^{NTugs} tugRel_m.Tugs<>_m$$

$$Tugs<[a,b] : L>_{NTugs} \stackrel{def}{=} \sum_{n=1}^{NTugs} tAcq_{j,n}.\ \overline{tGot_{j,n}}\ .Tugs<[a,b] : L>_{NTugs-n})$$

$$Tugs<[a,b] : L>_k \stackrel{def}{=} \sum_{n=1}^{NTugs} tAcq_{j,n}.Tugs<[a,b] : L :[j,n]>$$

$$+ \quad tRel_b.\ \overline{tGot_{a,b}}\ .Tugs<L>_k$$

$$+ \quad \sum_{m=1}^{b-1} tRel_m.(Tugs<[a,b] : L>_{k+m})$$

$$+ \quad \sum_{m=b+1}^{NTugs-k} tRel_m.(Tugs<[a,b] : L>_{k+m})$$

$$0<k<NTugs$$

$$Tugs<[a,b] : L>_0 \stackrel{def}{=} tRel_b.\ \overline{tGot_{a,b}}\ .Tugs<L>_k$$

$$+ \quad \sum_{m=1}^{b-1} tRel_m.Tugs<[a,b] : L>_k$$

$$+ \quad \sum_{m=b+1}^{NTugs} tRel_m.Tugs<[a,b] : L>_k$$

$$Jettys<>_{NJetts} \stackrel{def}{=} \sum_{n=1}^{NJetts} jAcq_{j,n}.\overline{jGot_{j,n}}.Jettys<>_{NJetts-n}$$

$$Jettys<>_{k} \stackrel{def}{=} \sum_{n=1}^{k} jAcq_{j,n}.\overline{jGot_{j,n}}.Jettys<>_{k-n} + \sum_{n=k+1}^{NJetts} jAcq_{j,n}.Jettys<[j,n]>_{k} + \sum_{m=1}^{NJetts-k} tugRel_{m}.Jettys<>_{(k+m)} \quad 0<k<NJetts$$

$$Jettys<>_{0} \stackrel{def}{=} \sum_{m=1}^{NJetts} tugRel_{m}.Jettys<>_{m}$$

$$Jettys<[a,b]:L>_{NJetts} \stackrel{def}{=} \sum_{n=1}^{NJetts} jAcq_{j,n}.\overline{jGot_{j,n}}.Jettys<[a,b]:L>_{NJetts-n}$$

$$Jettys<[a,b]:L>_{k} \stackrel{def}{=} \sum_{n=1}^{NJetts} jAcq_{j,n}.Jettys<[a,b]:L:[j,n]> + jRel_{b}.\overline{jGot_{a,b}}.Jettys<L>_{k} + \sum_{m=1}^{b-1} jRel_{m}.(Jettys<[a,b]:L>_{k+m}) + \sum_{m=b+1}^{NJetts-k} jRel_{m}.(Jettys<[a,b]:L>_{k+m}) \quad 0<k<NJetts$$

$$Jettys<[a,b]:L>_{0} \stackrel{def}{=} jRel_{b}.\overline{jGot_{a,b}}.Jettys<L>_{k} + \sum_{m=1}^{b-1} jRel_{m}.Jettys<[a,b]:L>_{k} + \sum_{m=b+1}^{NJetts} jRel_{m}.Jettys<[a,b]:L>_{k}$$

$$Model \stackrel{def}{=} (Tugs_2 \mid Jettys_2 \mid Boat \mid Boat \mid Boat) \setminus \mathbf{L}(Model)$$

Angle brackets denote lists of integer pairs, each consisting of a process tag and an amount required. Within lists capital letters denote sub-lists and square brackets contain one pair. ":" is the concatenation operator.

This result applies equally to the other blocking synchronisations, in particular to Bin and Store. Bin follows a similar, but unbounded, pattern to Res. Store requires a pair of lists, combining the double list structure of a CondQ with the pair list of the FIFO resource. In Chapter 6 some examples of CondQ and WaitQ models are shown, using this FIFO approach and experiments on these are given in Appendix C.

3.6 Further work

There are still a number of problems still to be resolved before a complete definition of simulation behaviour is reached. Most importantly, the analysis must be extended to consider stochastic models with continuous time. This is most obvious in the case of the definition of Interrupt. It requires considerable work, but Hillston's work with PEPA [37] and Strulo's CCS extensions [96] offer directions to consider. It may also be sensible to revisit the Synchronous Calculus and see if it offer solutions. Whatever solution is found needs to address the difficulties of generalising functional properties over ranges of timing and branching probabilities, which represent the environment and data dependent aspects of models.

Chapter 4

Graphical formalism for simulation

4.1 Introduction

This chapter defines the graphical formalism which formed the starting point of the work of this dissertation. The thesis being tested throughout this work is that it is possible to define formally a means of describing discrete event simulation models, to represent these as diagrams and to generate from these versions which can be solved for their quantitative properties, initially by simulation, and which can be used to prove useful results about their behaviour without resorting to simulation. In Chapter 2 a survey of typical graphical description approaches was presented. Here, a version suitable for the purposes of this dissertation is defined. This is built from those elements which were given formally defined semantics (in terms of CCS) in Chapter 3 and will be shown, in Chapter 5, to be capable of automatic translation into both a discrete event simulation language (*modified* DEMOS) and CCS, by constructing a tool which performs the task.

The starting point for the graphical formalism developed here is the activity diagram notation, introduced by Birtwistle [13] and extended slightly by Hughes [40]. The initial reason for this choice was familiarity with it and the availability of the DEMOS discrete event simulation package, for which it was developed. Indeed, the original purpose of the work leading to this dissertation was the production of a comprehensive graphical interface for DEMOS, but this was eventually relegated to a sub-task. As well as familiarity, activity diagrams were attractive because experience had shown them to be powerful as a description tool and intuitively simple to grasp.

As well as Birtwistle's and Hughes' work, a number of attempts have been made to produce activity diagram based graphical tools for DEMOS. The Process Interaction Tool developed at Edinburgh within the SIMMER Alvey project [72] led directly to

the work of this dissertation. Work in the SIMMER project also led to a first attempt to define a complete notion of activity diagrams [73]. The Process Interaction Tool also formed the basis of the PIT [6] work of the IMSE ESPRIT II project [75], where a number of concepts expressible in the vocabulary of activity diagrams, such as *servers* and *sources* of entities, were added by Uppal and Barber for perceived modelling convenience.

The contribution made here is to define carefully a minimum set of mechanisms which retain the generality of process based simulation modelling and, most importantly, a proper notion of hierarchical modelling, which is consistent with general rules for data abstraction and which is able to be mapped directly onto an underlying simulation language. In defining this set of mechanisms, first Birtwistle's activity diagrams are extended in line with Chapter 3 and then hierarchical modelling is considered, in search of completeness of description. A formal notation for describing such diagrams is created, using Extended Backus-Naur Form as its basis.

4.2 Extending activity diagrams for flat models

A set of diagrams to specify process based discrete event models is presented below. The approach developed is based on the informal conventions of activity diagrams first used to describe models for the DEMOS package, but here extended to allow complete descriptions of a much wider range of models. The set of mechanisms is that defined formally in Chapter 3. It forms basis of the concept of an *atomic* process in section 3.2. Descriptions at this level give the behaviour of a process in algorithmic terms, as a life cycle script.

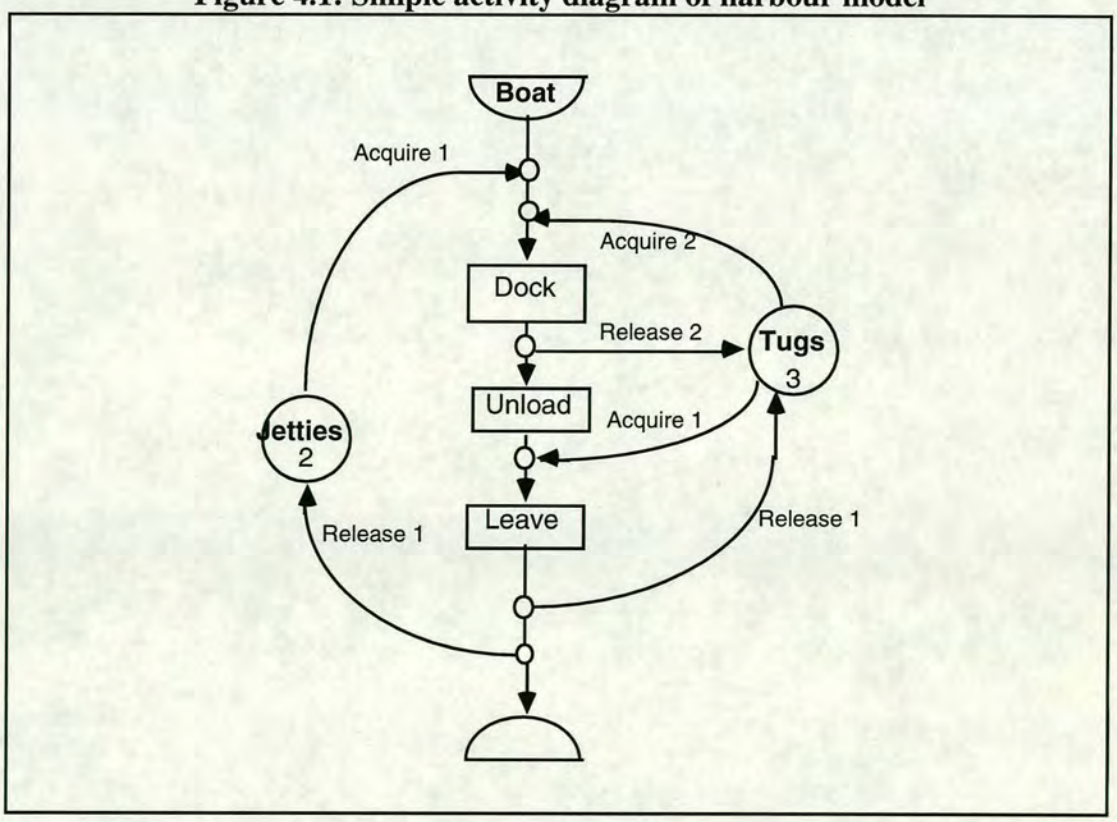
Graphical description of a process type requires both a way of showing the flow of control through such a process type and a way of representing interactions and synchronisations engaged in by instances of it. Construction of a model or sub-model defines the linkages between instances of processes, by mapping their required interactions onto instances of those objects which support such interactions. Many synchronisations among processes can be mapped onto queues, which is the only mechanism in queueing network based formalisms such as PAWS. However, the use of higher level abstractions, such as resources in GPSS, adds to the ease of description and widens the range of mechanisms which can conveniently be represented. Activity diagrams were defined to provide a convenient flow of control

description, based on flow charts, and to allow easy description of a wide range of useful synchronisation mechanisms, based on activity cycle diagrams. This makes them a good starting point for building a complete diagramming convention for process interaction.

4.2.1 The model from chapter 2 again

A simple example of an atomic process description is shown in Figure 4.1. The model is the harbour from Chapter 2, which was examined in CCS in section 3.5.1. It includes Birtwistle's standard symbols of a rectangular box for a delay, annotated with a description of the associated activity, and a circle for a resource, annotated with a description of the resource and the initial amount available. New symbols are needed to complete even this simple example. Hughes added a lower semi-circle, annotated with the process name, which marks the start of the process life cycle, and an inverted form of the start symbol, with no annotation, to mark the termination of the process. In the Simmer Process Interaction Tool synchronisation nodes were also added, to show where resources are acquired and released. This last extension is a significant change from Birtwistle's convention of attaching synchronisations to hold boxes and allows the exact order of all such synchronisations to be specified.

Figure 4.1: Simple activity diagram of harbour model



Various forms of arrowed line could be used to represent the type of a link, but the actual type is fully determined by the types of the nodes which it joins. Thus the lines joining delay to delay, delay to start or delay to termination represent control flow in the process, in the same manner as in conventional flow charts. On the other hand, the lines joining resources to synchronisation nodes represent acquisition or release of amounts of those resources.

Acquisition and release constitute, respectively, a potential blocking of the flow of control in the process due to contention with other processes and a potential freeing of another process currently blocked by this process. The amount to be acquired or released is shown as an annotation to the link, while the direction of the arrow on the line determines which action is intended. All external interactions are shown by synchronisation nodes. In this sort of process type description the objects to which synchronisation nodes are linked are there purely to show the type of synchronisation by which any instance of this type will be linked to other process instances. As it happens, this example does not use other process types, simply a stream of Boat processes. In such simple cases the model can be completely described by suitable annotation of the process type description, with amounts of resources and inter-arrival times added in this case. This is analogous to very simple computer programs, where procedural abstraction is not needed

4.2.2 The complete menu of symbols

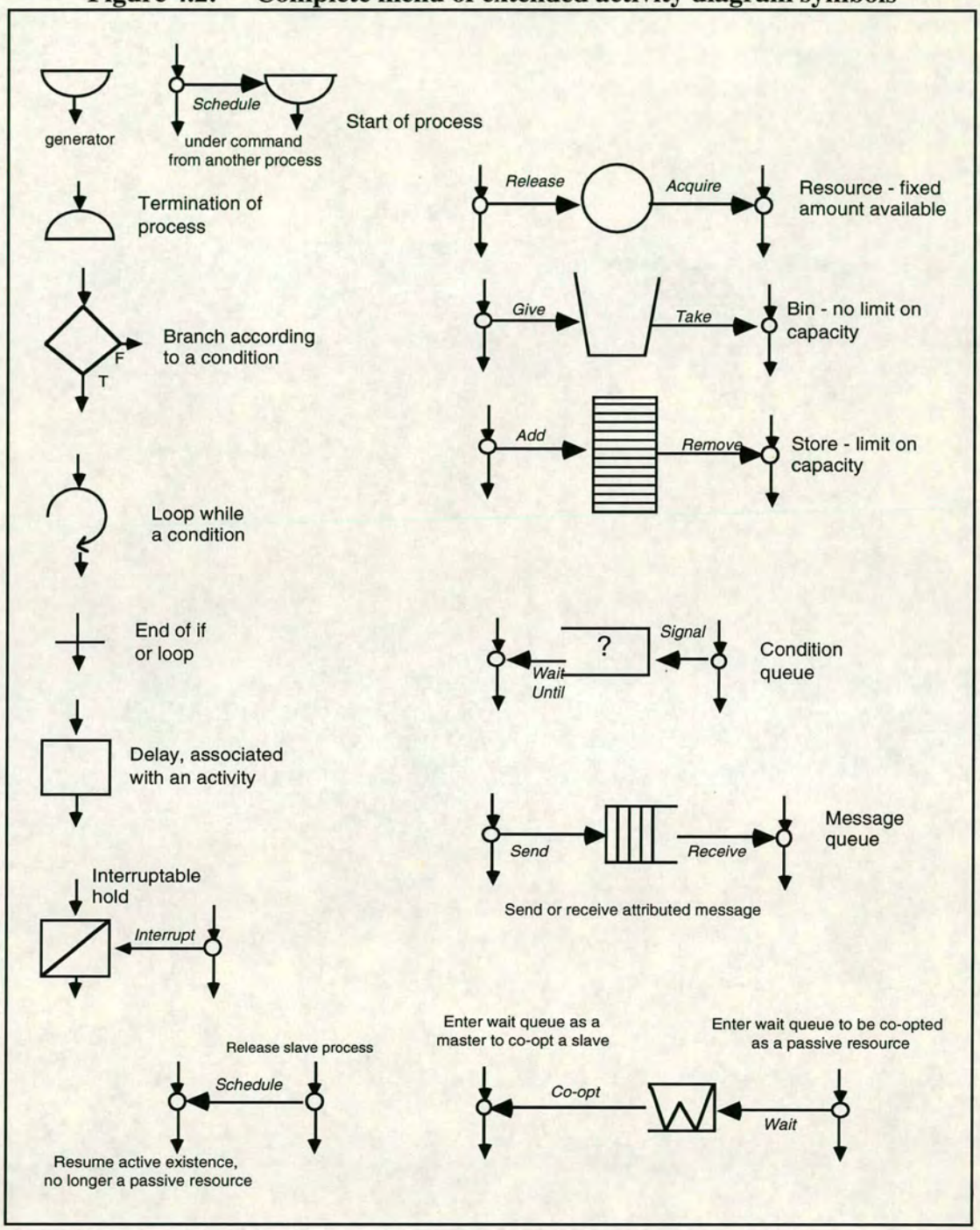
Figure 4.2 shows the complete set of symbols used in extended activity diagrams to describe atomic processes. These are divided into flow of control symbols and synchronisation symbols, involving resource and queue blocking.

4.2.3 Flow of control symbols

The flow of control symbols are similar to those used in conventional flow charts, with *decision* nodes, *loop-start* nodes, *branch/loop-end* nodes, *start* and *terminate* nodes. There are also *hold* nodes, which represent activities whose durations are defined by expressions containing constants, visible state variables and stochastic variables. Holds are usually regarded as part of the flow of control, but this is considered in more detail in section 4.2.5 below. *Synchronisation nodes* indicate points at which the flow of control requires an interaction with another process instance. These nodes are linked by directed edges indicating flow of control. There must be a connected path from the start node to all other nodes and from each other

node to the end node. The algorithmic description of an atomic process type is contained in the directed graph made up from these symbols.

Figure 4.2: Complete menu of extended activity diagram symbols



The presence of the loop-start/end nodes removes the need for cycles in these graphs. A decision or loop-start node is associated with the next succeeding branch/loop-end node.

4.2.4 Synchronisation and communication primitives

The second set is of symbols which describe interactions between process instances. Two forms are used; links direct from one process to another and links to passive objects.

> *Direct scheduling* of one process by another is shown as an arrow from one synchronisation node to another or, where creation as well as scheduling is implied, to a start node. *Interruption* of an activity by another process as an arrow from a synchronisation node in one process into a hold in another.
>
> Communication through a *passive object*, such as a resource or a condition queue, is shown by an outgoing arrow to the passive object from a synchronisation node in the output process and an incoming arrow from the passive object to a synchronisation node in the input process.

The distinction is actually more a descriptive convenience than a necessity, as discussed in section 4.2.5 below. From a syntactic point of view, these conventions allow unambiguous identification of any *synchronisation node/directed edge/second node* triple. Semantically synchronisation nodes are ambiguous, as are edges, but their meaning is always established by the syntactic triples in which they must be found.

The *wait queue* is a double queue. One, slave, process signals that it wants to become a passive, attributed object. Another, master, process requests from this queue a coopted slave process which remains passive until it is rescheduled, by a subsequent direct scheduling.

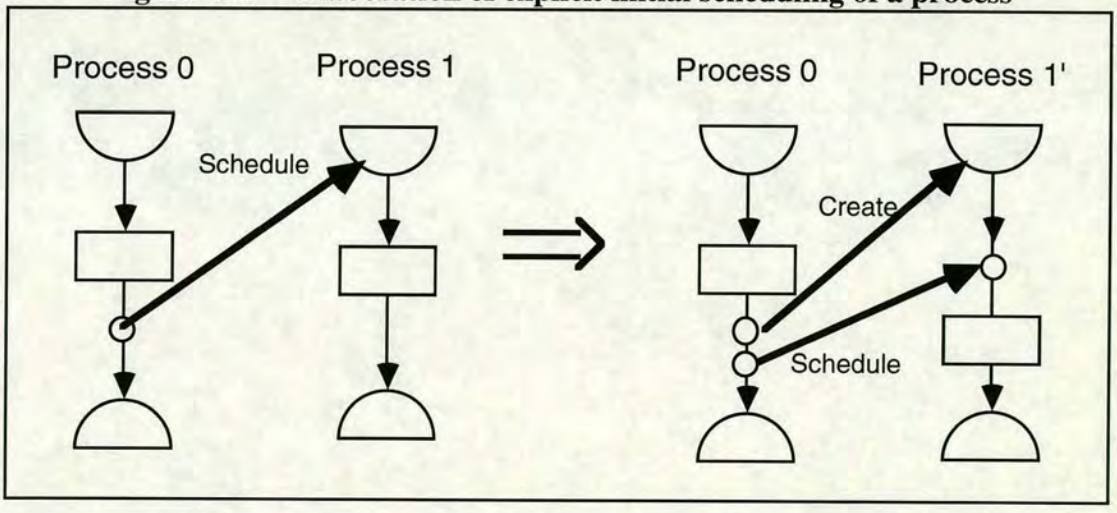
Several communications are attributed or parameterised. A resource, store or bin request has an amount, a condition queue request has a Boolean expression and a message queue request has an object. Wait queue communications have processes as attributes.

4.2.5 A digression on holds and schedules

This section picks up certain loose ends concerning the representation of the passage of time in process based simulation models. In section 3.3.2 it was noted that sequential behaviour of processes can be broken into sub-processes which schedule each other. In the description of flow of control above it was noted that direct scheduling of one process by another has a slightly different representation to other interactions between processes. It is possible to resolve these points in the context of extended activity diagrams, but the result is slightly more cumbersome.

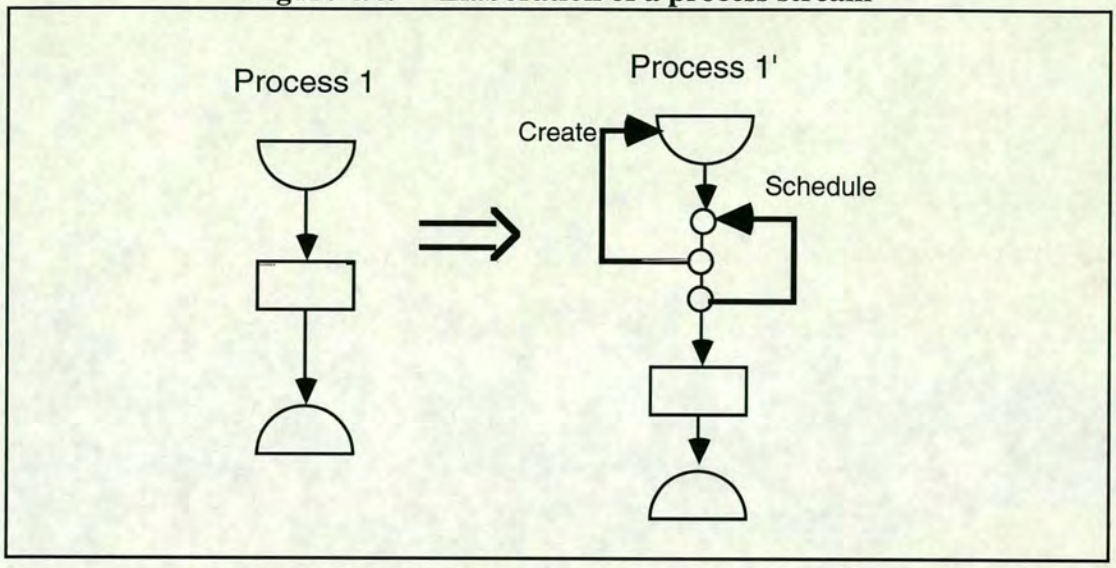
The first point is that the instantiation and scheduling of a new process, currently represented by an arrow from a synchronisation node into the start node of another process could, for consistency, but at the expense of more nodes, be represented as shown in Figure 4.3. This implies that any newly created process is initially passive, until explicitly scheduled. This is in fact what happens in the equivalent DEMOS code.

Figure 4.3: Elaboration of explicit initial scheduling of a process



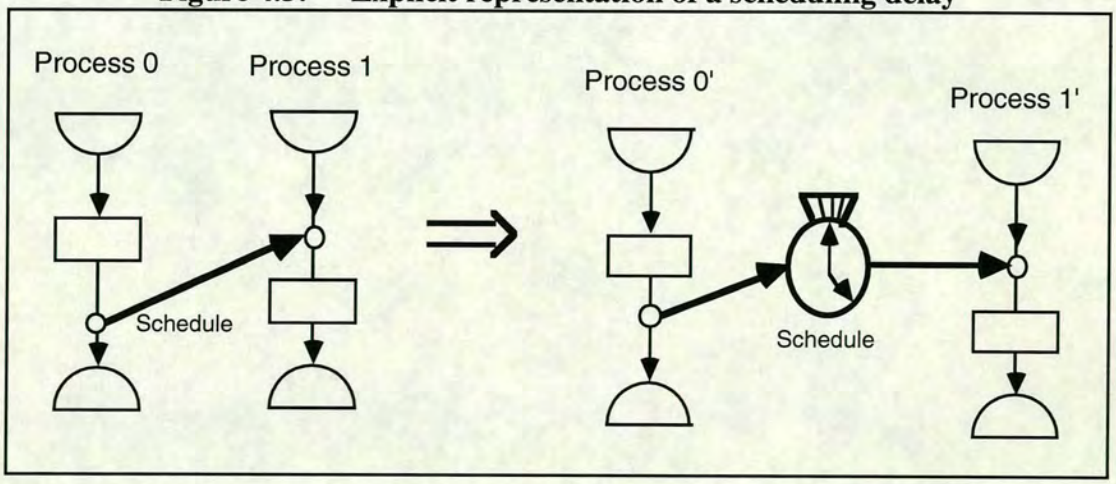
A second elaboration is to force an explicit representation of the scheduling of a stream of process instances, often modelled in DEMOS as each instance first creating and scheduling a successor before beginning its own activities. Figure 4.4 shows the extended activity diagram view of this. In STC's version of PIT this was handled by defining a special process, called a Source, which cycled endlessly scheduling a new instance of the process in the stream and then holding for the inter-arrival time.

Figure 4.4: Elaboration of a process stream



Thirdly, it is possible to remove the special structure of one process scheduling another, by introducing a time delay node between the scheduler and the scheduled. This treats time as a state variable like any other and its advance as potentially unblocking a delayed process. Although this is a realistic approximation to the underlying event list mechanism, it requires a rather low level view of the model from the modeller's point of view. However, it does promote consistency in the representation of state change. Figure 4.5 shows the effect on a diagram.

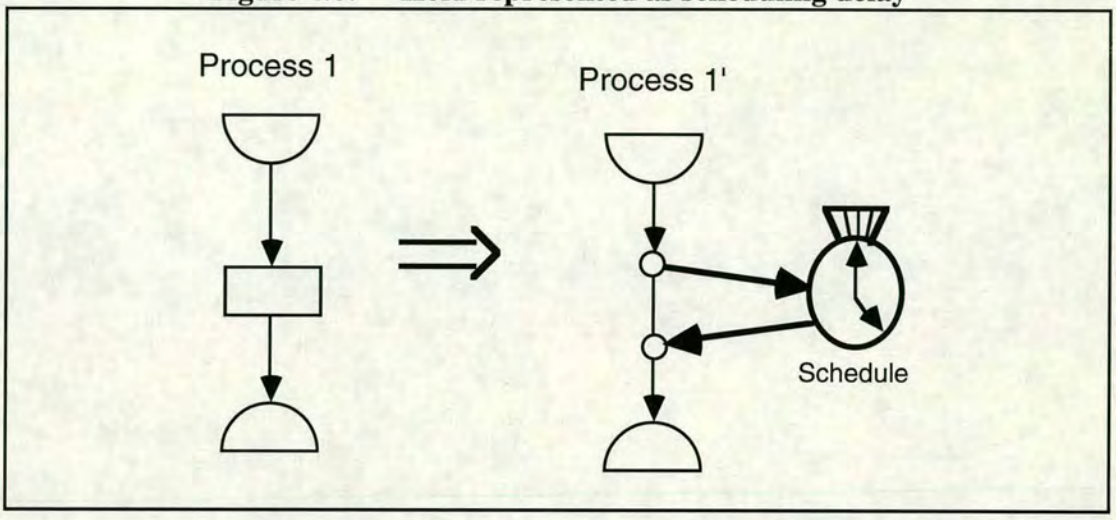
Figure 4.5: Explicit representation of a scheduling delay



Finally, the discussion of section 3.3.2 can be applied, either using the explicit delay in scheduling or not, to remove Hold as a flow of control symbol and unify the

notion of time delay and the other forms of blocking in a model. Essentially this results in a hold becoming two synchronisation nodes, from the first of which an arrow goes to a scheduling delay node and into the second of which an arrow returns from that node. Figure 4.6 shows this effect, in a similar manner to Figure 3.3.

Figure 4.6: Hold represented as scheduling delay



The result could be thought of as a canonical set of symbols for the representation of process based simulation models. In the notation actually used in this dissertation the time advance aspects are abstracted from the general state change concept and holds are used, along with direct scheduling between processes.

4.2.6 A formal grammar for extended activity diagrams

The diagrams presented are in fact a formal language. Like any textual language it is desirable to be able to express the syntactic structure of extended activity diagrams through a suitable grammar. Unlike textual languages, diagrams are two dimensional. This requires a slightly extended form of the Backus-Naur type meta-languages normally used for expressing context free programming languages.

Figure 4.7: Grammar of flat level of extended activity diagrams

graph = *object* *

object = **res** | **bin** | **store** | **condq**
| **waitq** | **messageq** | *process*

process = **start** *thread* **end**

thread = *flowcom* *

flowcom = *syntriple* | *cond* | *loop* | **hold**

cond = **if** < *thread* || ((**llink** *thread* **rlink**) | (**rlink** *thread* **llink**)) > **end**

loop = **while** *thread* **end**

syntriple = *acquire* | *take* | *remove*
| *receive* | *waituntil* | *coopt* | *newsched*
| *release* | *give* | *add* | *send*
| *signal* | *wait* | *schedule* | *interrupt*

acquire = **synch inlink res**

take = **synch inlink bin**

remove = **synch inlink store**

receive = **synch inlink messageq**

waituntil = **synch inlink condq**

coopt = **synch inlink waitq**

newsched = **synch outlink start**

release = **synch outlink res**

give = **synch outlink bin**

add = **synch outlink store**

send = **synch outlink messageq**

signal = **synch outlink condq**

wait = **synch outlink waitq**

schedule = **synch outlink synch**

interrupt = **synch outlink hold**

The structure of extended activity diagrams is almost that vertical connection implies flow of control and horizontal connection implies communication/synchronisation. In fact, apart from horizontal branching to distinguish false from true outcomes in decisions, this is always true. If it is examined more closely, the brief horizontal divergence after a decision can be taken as an elided instantaneous scheduling or more simply as a splitting into two vertical continuations. In either interpretation it can be ignored. The *top to bottom* dimension is more complex than the horizontal, since it contains nested structures - loops and conditional branches - enclosed in bracketing symbol pairs. The horizontal dimension of communication synchronisation is expressible in a very simple regular expression grammar, while the vertical dimension of flow of control requires a more general context free grammar. By treating these two dimensions as distinct, as if links in one have a different significance to those in the other, and by distinguishing two types of horizontal link - *incoming* and *outgoing* - and two types of vertical link - *downward* and *branching*, a complete meta-language and grammar can be defined.

In this grammar, normal extended BNF conventions are followed quite closely. Bold face is used for terminal symbols, italics for non-terminals, Times Roman for meta-symbols. The normal BNF meta symbols used are vertical bar ("|") for alternatives in a production, equals ("=") for production, asterisk ("*") for repetition of symbols ("one or more"), parentheses to delineate sub-expressions. In an extension of BNF additional symbols are used; angle brackets to delineate forking ("<") and joining (">") of parallel vertical sequences and double vertical bar ("||") to separate such sequences.

Unlike normal BNF grammars, this produces diagrams composed of linked nodes, rather than strings of characters. This requires a modified understanding of juxtaposition of symbols. Unless modified explicitly, wherever one symbol follows another it should be taken as meaning that the first symbol either occurs directly above the second or is linked to it by an arrow which leaves the bottom of the first and enters the top of the second. The angle bracket / double bar notation is an explicit indication that the parallel sequences defined share a common preceding and succeeding node. The other explicit modifiers are **llink** (**rlink**), which says the first symbol is linked to its successor by an arrow leaving its left (right) side and entering the right (left) side of its successor, and **inlink** (**outlink**), which says that the first symbol is linked to its successor by an arrow into (from) the first, from (into) the

second. The route taken by these links and the exact position of the nodes is not fixed, allowing them to be drawn in an infinite number of ways, but guaranteeing the connectivity of the resulting graph.

4.3 Typical examples of extended activity diagrams

There follow two further examples of practical, flat models, represented as extended activity diagrams. They show that a single level of description can be sufficient for modelling, but also approach the limit of what can be described without resort to some form of hiding of detail.

4.3.1 A simple example

To see the use of a selection of these symbols, consider Figure 4.8, which is a practical example, described more fully in [68]. This contains all the typical elements in a single level activity diagram representation of a model.

The model represents a lineprinter connected to several host computers on a network. Each host process has a life cycle in terms of the lineprinter. The other activities of the Host are ignored in this model, but they could quite easily be reflected stochastically in the inter-polling time. A Host tries to gain access to the lineprinter whenever it has a file in its print queue. If it is unsuccessful, it will back off for some inter-poll time and then try again. If it is successful, it seizes the lineprinter until it has printed its file. It is then required to back off for a longer time, before trying again. This is designed to allow other machines to achieve access more easily.

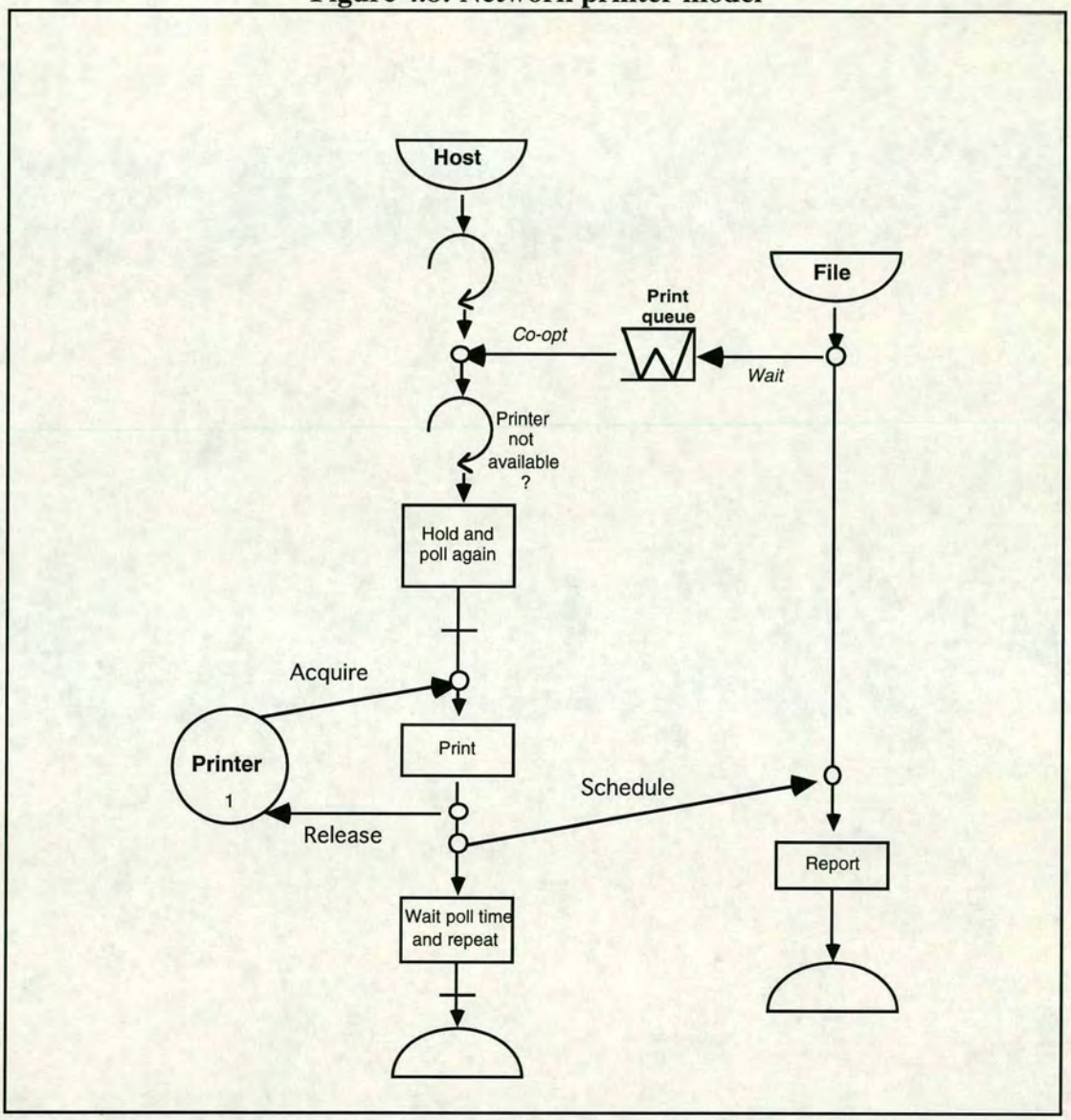
The practicality of the scheme being modelled is not really important, although it matches a genuine design. What is useful is that the model demonstrates many of the symbols in the vocabulary above. Two important additions to the set used in the first model are to be seen:

loop-start and end nodes, annotated with a condition and showing forking and joining;

master/slave processes; allowing one process to act as a passive resource to another for part of its life cycle.

Note that the host process never terminates. Files start as active processes, allowing each to schedule its successor and thus determining the inter-arrival rate of files. They then enter the Host's printer queue, becoming passive objects. They are finally reawakened by the Host after printing and terminate after reporting.

Figure 4.8: Network printer model



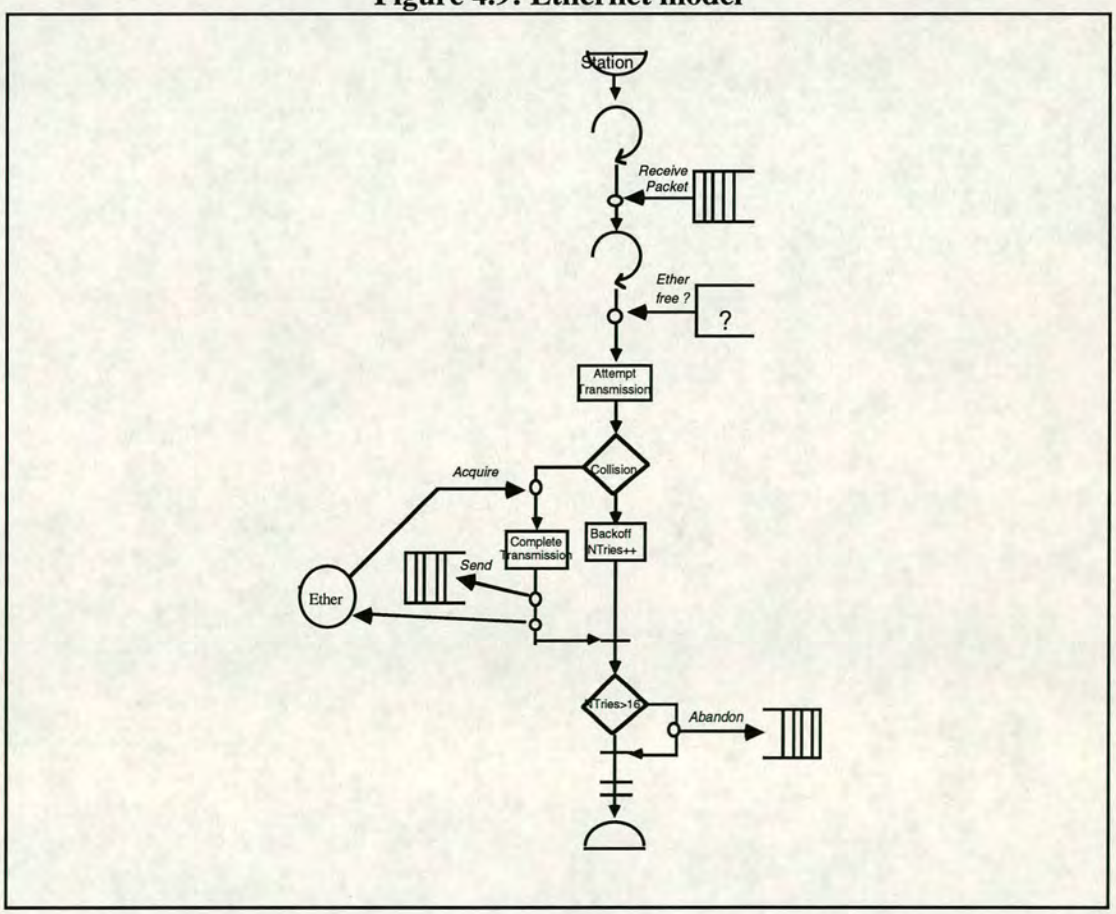
4.3.2 A further practical example

Figure 4.9 illustrates the use of the condition queue, by means of an apparently simple model of an ethernet like protocol. The CSMA property of ethernets is that no station may attempt to transmit while the channel is busy, i.e. another station is

broadcasting. This is represented in the diagram by a conditional wait following the *attempt transmission* phase of the transmitter's life cycle. This delays the process until the condition is satisfied. The use of such a device in the diagram makes the model rather simple to describe. Unfortunately such a feature is notoriously difficult to program. The required effect is that once the channel is freed by its current user all the transmitters waiting for it try to transmit simultaneously. This effect of simultaneity is not natural to the interleaved execution of most process based simulation systems. A solution to this problem is considered in depth in Chapter 6.

In this example the packets are passive, but possess attributes, such as length, and so are drawn as arriving in message queues. The diagram is clearly incomplete as a description of a model, since there is no indication of how packets arrive or are disposed of. In fact it is a useful working sketch of the behaviour of one part of a model, but is not sufficient as a description of that model. The complete model would be too complex to fit easily as an activity diagram. Having reached roughly the limits of activity diagrams, a more extensive approach is required to continue.

Figure 4.9: Ethernet model



4.4 Hierarchy - Configuration Diagrams

In a further refinement to activity diagrams, hierarchical modelling of compound processes in terms of their constituent sub-processes or components is now allowed. This is expressed in the form of configuration diagrams, which are introduced here.

The use of a diagramming technique has the beneficial effects of:

natural expression of parallelism;

encouragement of high level thinking;

easier interchange of ideas with non-programmers.

The use of diagrams leads to the formulation of small models, which is generally a good thing. Too often the tendency is to over-model. However, in some cases there is a need to model quite complex systems in more detail than can be represented in a single diagram. Figures 4.8 and 4.9 are probably as complicated as is sensible for the paradigm of activity diagrams.

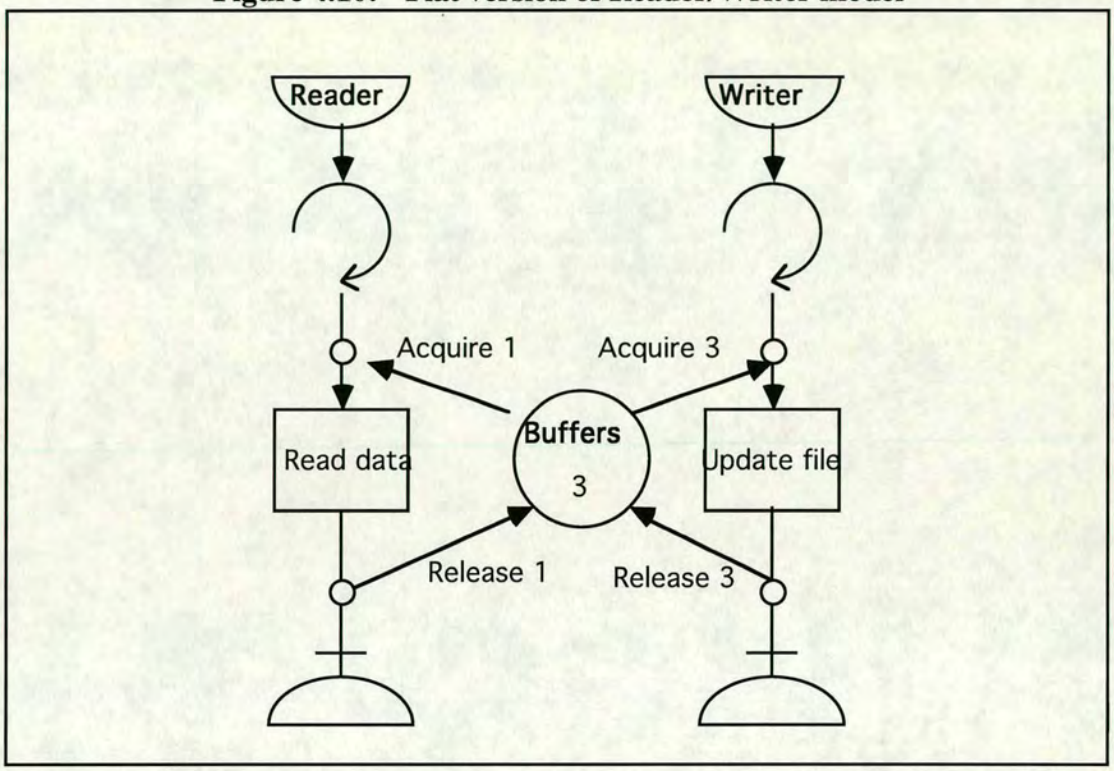
As defined in Chapter 3, a compound process consists of a number of instances of interacting sub-processes and their synchronisation mechanisms. This approach allows partitioning of a model into sub-models hierarchically, since a non-atomic sub-process at one level can itself be decomposed into further sub-processes. An important benefit arises from the fact that many real world systems are structured in an analogous way and so this approach allows the structure of the real world system to be retained in the simulation model. This overall approach is equivalent to object oriented programming, with each process description equivalent to a class and each process instance equivalent to an object. Thus the realisation of such models in an object oriented language proves very straightforward. SIMULA [12,74] is almost ideal in this sense.

4.4.1 A simple hierarchical model

Figure 4.10 is taken from example 4.1 of [13] and shows a model where two processes co-exist. Each is sufficiently simple that no confusion or crowding results from combining them. However, it is easy to imagine that more detailed modelling

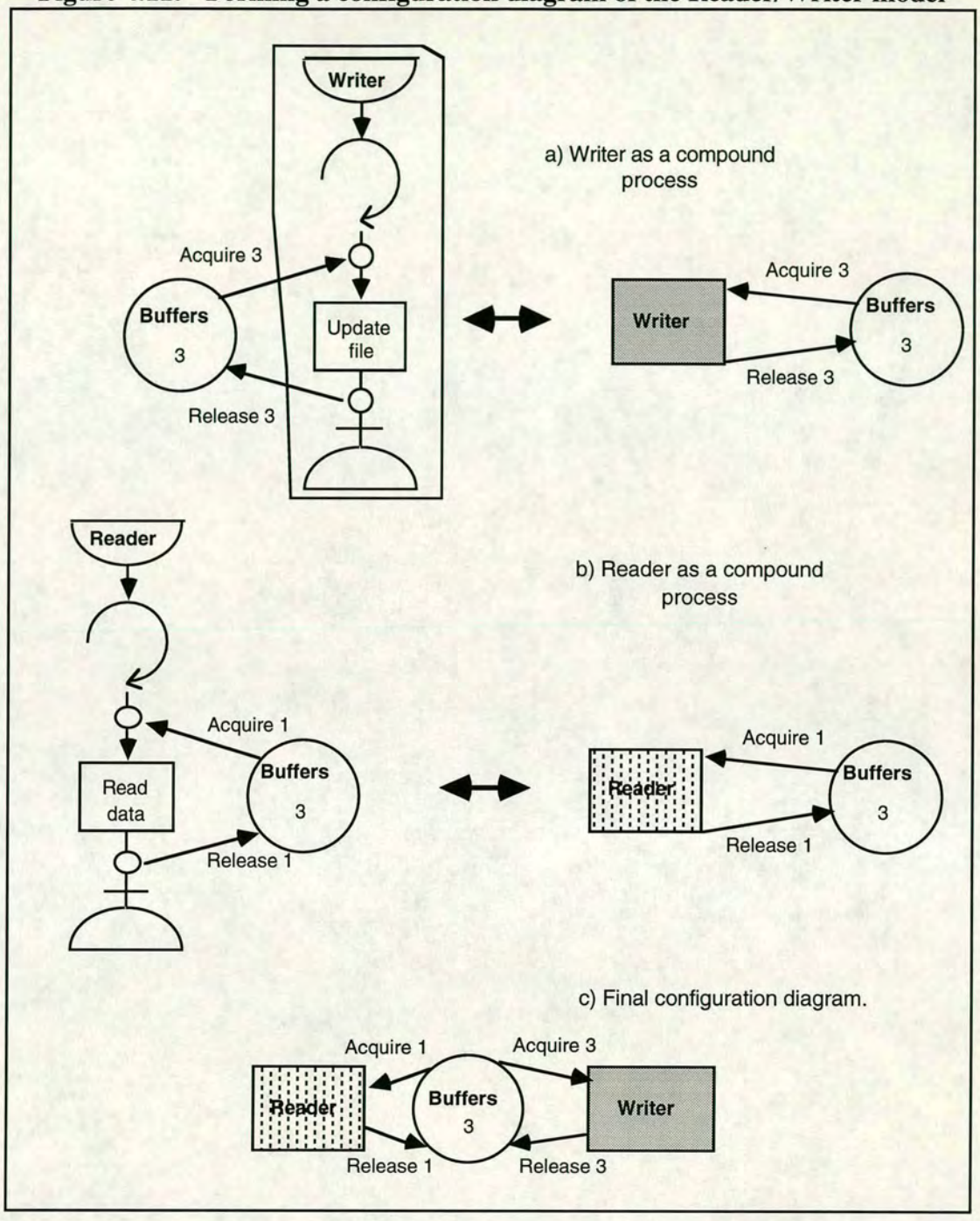
of the individual processes, essentially expanding the hold boxes '*read data*' and '*write data*' into full algorithmic descriptions, could make the diagram unreadable. Then some means of hiding detail becomes necessary. Ideally this should match some real property of modularity in the system represented by the model. The technique suggested is to use *configuration diagrams*, as shown in figure 4.11.

Figure 4.10: Flat version of Reader/Writer model



The algorithmic detail of the atomic process descriptions, contained in the activity diagrams for reader and writer processes, is suppressed, leaving only the external links to the processes visible (figures 4.11a and 4.11b). Module level description allows more complex systems to be described, without overcrowding the diagram. It also has other advantages, as example 4.12 will show.

Figure 4.11: Forming a configuration diagram of the Reader/Writer model



By combining the two modularised atomic processes, a compound process description or model description (Figure 4.11c) is produced, depending on the level in the model. Both are represented by configuration diagrams. Here diagrams only use a simple box and appropriate link symbols from activity diagrams, such as the resource, bin and queue symbols. Only those links which can be used to attach this

component to others are represented here, but in fact there is no reason not to mix atomic and modularised processes. In the tool described in Chapter 5 more restricted conventions are needed, as outlined below.

4.4.2 A practical example using hierarchy

Figure 4.12 shows a very complex activity diagram, containing two processes, the PINP (packet input process) and the POUTP (packet output process) of an X.25 type protocol, level 3. For details of models of the full protocol see Pooley and Birtwistle[69] and Belsnes and Bringrud [11]. This simplified version is actually very similar in structure to the example of figure 4.10, but the description is far more detailed. This makes it a candidate for the use of configuration diagrams to allow further modelling without sacrificing readability.

Figure 4.12: Flat model of level 3 of X.25 type protocol

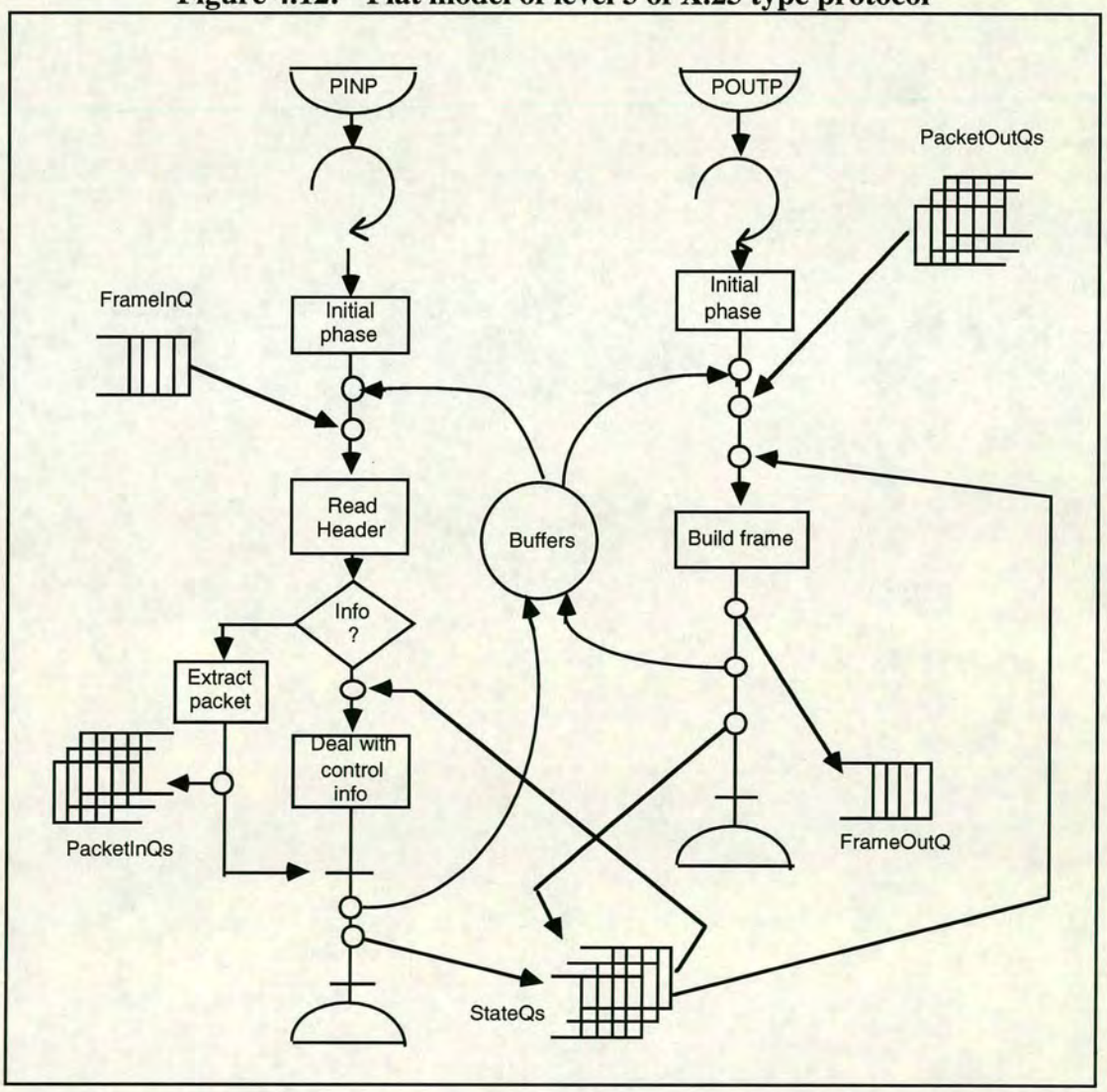


Figure 4.13 shows the process of turning the model into a configuration diagram, which is a process of *abstraction*. Note that the modularisation of the lower level description matches the logical and physical structure of the system modelled. This is a natural and good use of abstraction.

Figure 4.13: Module abstraction of X.25 level 3

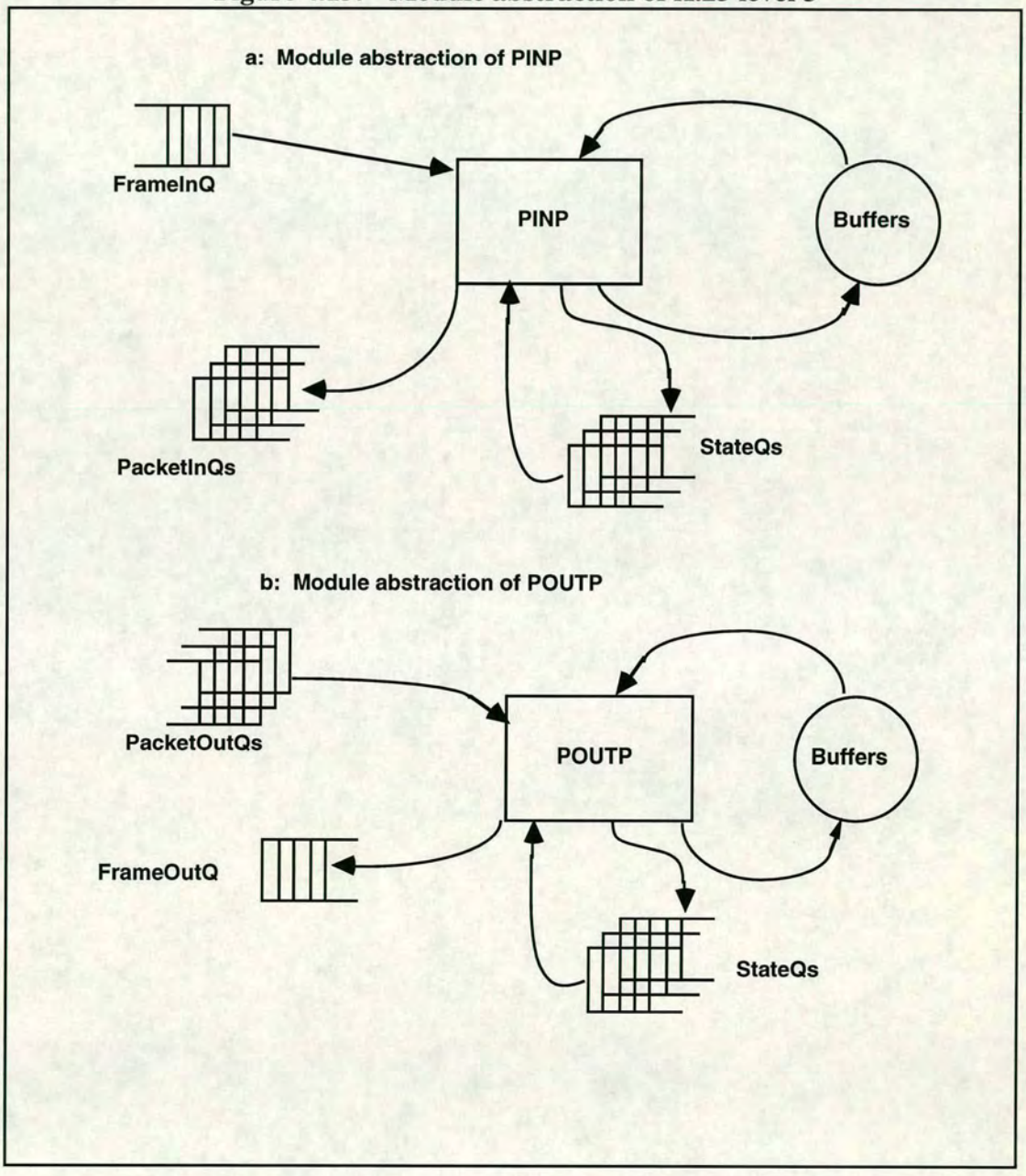
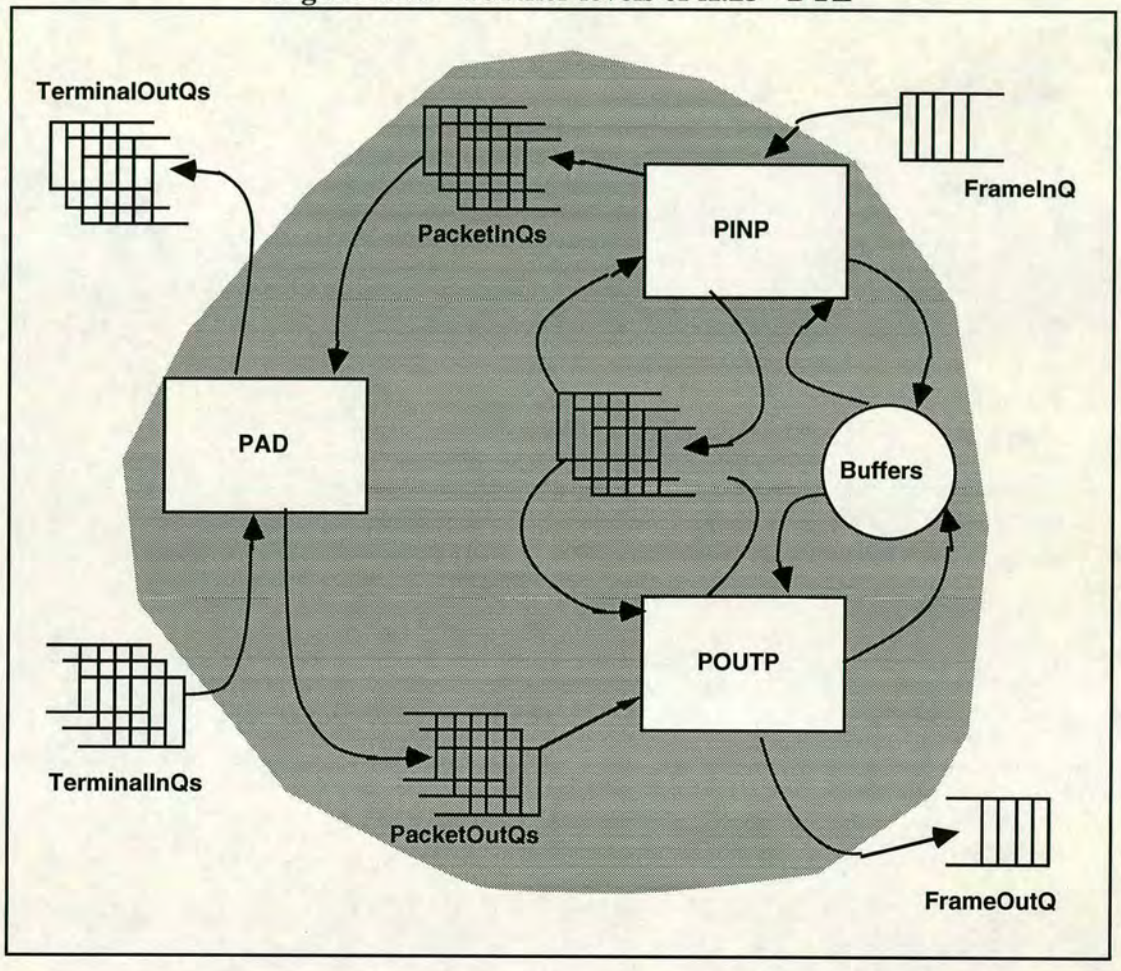


Figure 4.14 shows the use of the resulting process descriptions, along with a module level description of the PAD (packet assembler/dis-assembler) process to describe a

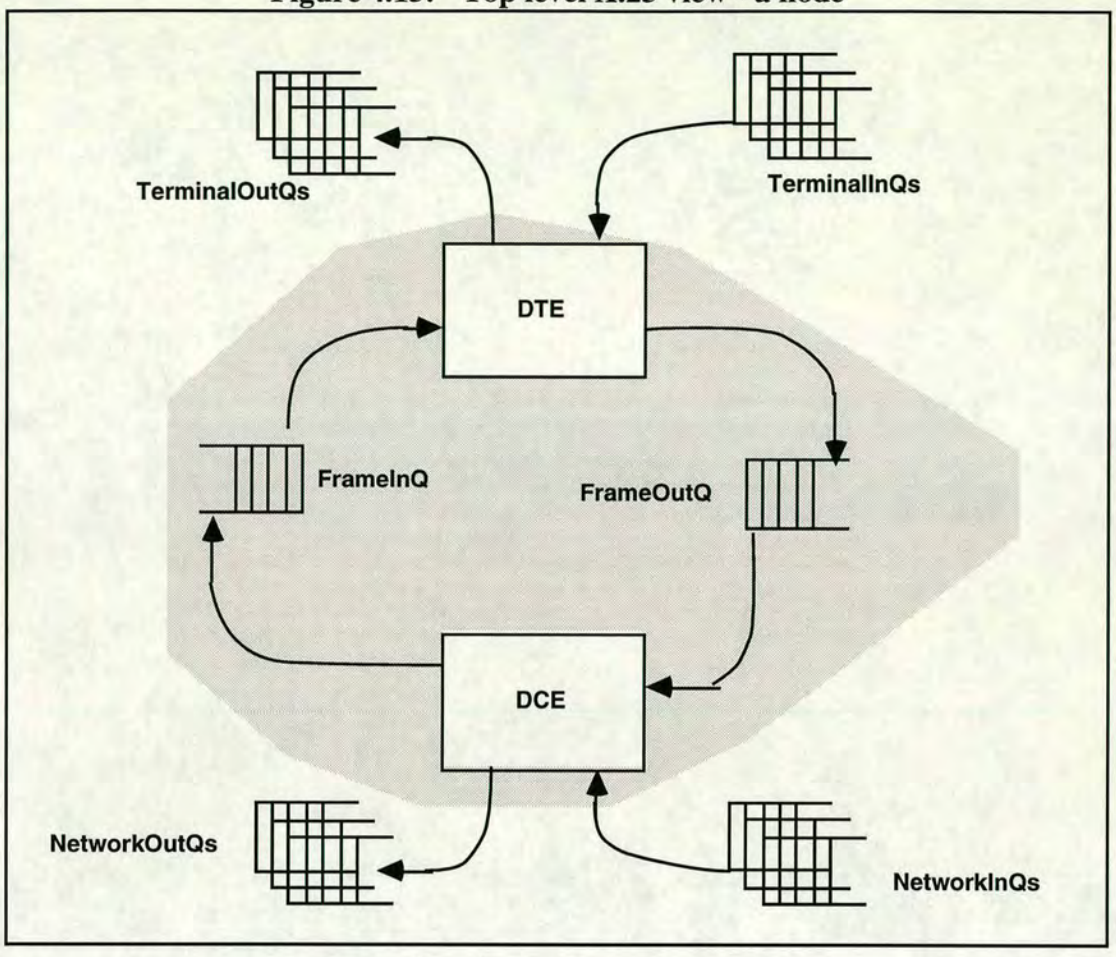
compound DTE (data terminating equipment) process. Each compound process description or process module description preserves its external links, but hides internal detail. This is an object oriented modelling view, where only the external interface to an object is accessible to other objects.

Figure 4.14: Further levels of X.25 - DTE



Finally, figure 4.15 shows the recursive application of configuration diagrams, with the DTE process being reduced to a single compound process and then combined to form a node compound process description. Such abstractions are applicable in theory to arbitrary depths of description, allowing correspondingly complex systems to be described.

Figure 4.15: Top level X.25 view - a node



4.4.4 Grammar and types for configuration diagrams

Certain safeguards must be applied when combining modules in this way. Conceptually, it is merely necessary to overlay equivalent links. For this to have true meaning, however, the links must be of equivalent type. The convention demands the notion of *strong typing* for all components and links. The type of a communication/synchronisation link is defined by the object at the other end of it, which is expressed syntactically in the grammar of extended activity diagrams in section 4.2.7. For configuration diagrams, this grammar is further extended by adding a new alternative for *object*, called **submodel** and allowing this, followed by a number of **synch** nodes corresponding to the number of external links (parameters) and schedule points for this modularised process and a number of **hold** nodes, corresponding to interruptable holds visible within this modularised process, to form a *subprocess*. Only one link is needed from a *subprocess* to an object, even if there would be more than one in its atomic level description, since it is merely a reference

to the object which is the actual parameter to match a formal parameter in the DEMOS Entity. In the CCS model it is the name by which an internal action is relabelled. In principle the link could be in either direction, but here it is assumed to be *from* the *subprocess to* the other object. This may seem to lose information which might be important, but the tool in Chapter 5 demonstrates that it is sufficient. Links to **submodel** nodes are assumed to be *newsched* triples. Links to the **synch** nodes of a *subprocess* are assumed to be *schedule* triples. Links to a hold node of a *subprocess* are assumed to be interrupt triples. The last two are problematic, as they require knowledge of the internal behaviour of the *subprocess* to be used correctly. They are included for completeness, but are not expected to be widely used.

Figure 4.16: Full grammar of extended activity diagrams

graph = *object* *

object = **res** | **bin** | **store** | **condq**
| **waitq** | **messageq** | *process* | subprocess

process = **start** *thread* **end**

thread = *flowcom* *

flowcom = *syntriple* | *cond* | *loop* | **hold**

cond = **if** < *thread* || ((**llink** *thread* **rlink**) | (**rlink** *thread* **llink**)) > **end**

loop = **while** *thread* **end**

syntriple = *acquire* | *take* | *remove* | *receive*
| *waituntil* | *coopt* | *newsched*
| *release* | *give* | *add* | *send*
| *signal* | *wait* | *schedule* | *interrupt*

acquire = **synch inlink res**

take = **synch inlink bin**

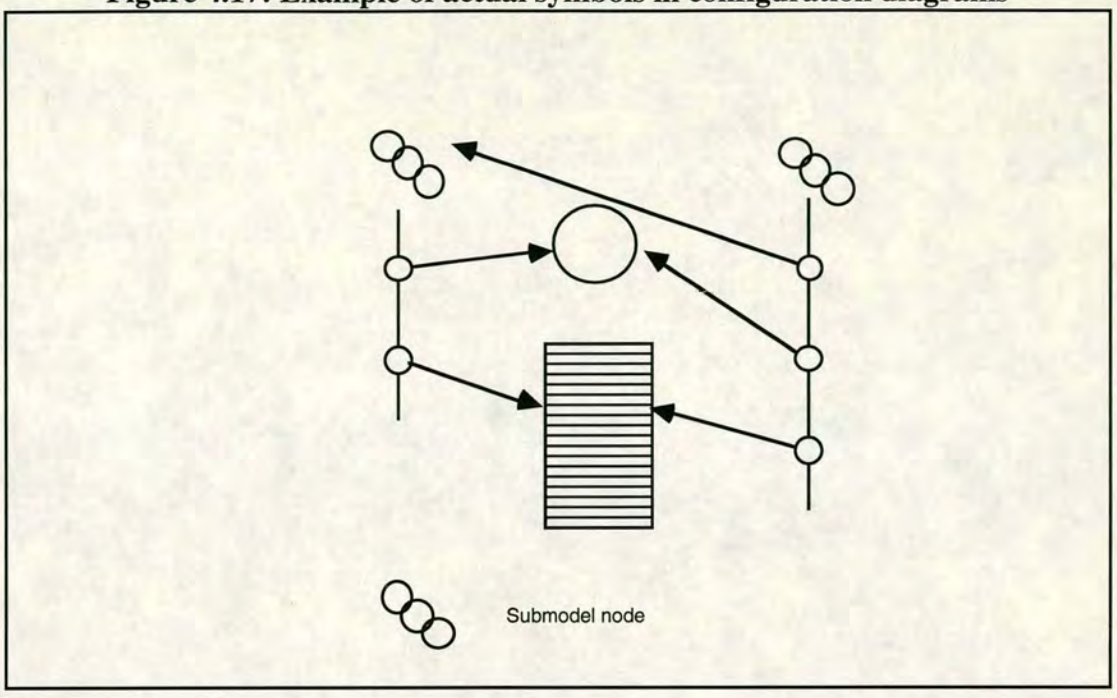
remove = **synch inlink store**

receive = **synch inlink messageq**

waituntil = **synch inlink condq**

coopt	=	**synch**	**inlink**	**waitq**				
newsched	=	**synch**	**outlink**	(	**start**	|	**submodel**	)
release	=	**synch**	**outlink**	**res**				
give	=	**synch**	**outlink**	**bin**				
add	=	**synch**	**outlink**	**store**				
send	=	**synch**	**outlink**	**messageq**				
signal	=	**synch**	**outlink**	**condq**				
wait	=	**synch**	**outlink**	**waitq**				
schedule	=	**synch**	**outlink**	**synch**				
interrupt	=	**synch**	**outlink**	**hold**				
subprocess	=	**submodel**	(*parbind*)*					
parbind	=	*resbind*	|	*binbind*	|	*storebind*	|	*mqbind*
	|	*cqbind*	|	*wqbind*	|	*entbind*		
resbind	=	**synch**	**outlink**	**res**				
binbind	=	**synch**	**outlink**	**bin**				
storebind	=	**synch**	**outlink**	**store**				
mqbind	=	**synch**	**outlink**	**messageq**				
cqbind	=	**synch**	**outlink**	**condq**				
wqbind	=	**synch**	**outlink**	**waitq**				
entbind	=	**synch**	**outlink**	**start**				

Figure 4.17: Example of actual symbols in configuration diagrams



4.4.5 Application specific description

In most models such abstractions will correspond to actual components. In many case studies only a small number of modules will need to be redefined, as many lower level ones will have remained unchanged from earlier work. This leads towards the notion of *reusable module definitions*. This applies both to the diagrams and to any other form of representation, including separately compiled object modules. In practice, most modelling of complex systems is probably able to work in terms of libraries of standard component models, with only a few additional or changed algorithmic descriptions. Thus, most modelling in a particular field will be able to proceed in terms of configuration diagrams alone. As this involves no knowledge of component implementation, it is expected to be a much more natural and attractive level for non-specialist modellers to use.

It is also possible to define meaningful graphical representations for process modules, resources and queues in diagrams, to enhance their readability for users from particular backgrounds. Thus, it might be claimed that the configuration diagramming technique is extensible towards particular application areas. In subjects such as modelling of flexible manufacturing systems (FMS), existing conventions can be incorporated removing the need to master a new set of symbols.

4.4.6 Top-down and bottom-up

The examples shown all proceed bottom-up, i.e. building from simple, low level components towards complex systems. This is merely for ease of explanation, starting by constructing single level, algorithmic descriptions. In fact model design may proceed top-down just as easily. Thus, it is possible to sketch the top level of a system as a configuration diagram and decompose this to give lower level component descriptions. The only requirement is that, for a complete system description, the higher level descriptions must all lead ultimately to an algorithmic description.

4.5 Conclusions

The technique of extended activity diagrams, including configuration diagrams offers a flexible way of describing models in process oriented terms. Such descriptions are very useful for communicating models both amongst modellers and to laymen. The set of symbols suggested here is believed to form the basis of a standard for such descriptions.

By supplying appropriate information about each symbol's attributes, it is also possible to provide sufficient information to make the coding of actual programs from these diagrams completely mechanical. This allows the use of direct graphical entry of simulation models on graphical workstations, thereby extending the concept of dialogues [14] to include graphics as input. In Chapter 5, a new version of such a tool is built for the purposes of this dissertation, using the concepts developed in this and the preceding chapter. It adds the important new capability of generating CCS equivalent models directly from the same representation.

Activity and configuration diagrams are, it is contended, an important step away from the need to view the mastery of programming as a key part of effective simulation, by allowing the modeller to concentrate on understanding the modelling process and so removing a major barrier to more widespread use of simulation.

Chapter 5

A tool to demonstrate and simplify combined modelling

5.1 Introduction

In Chapter 3 the possibility of defining a mapping between CCS descriptions of behaviour and an extended version of the DEMOS simulation language, *modified* DEMOS, was explored. In Chapter 4 a graphical formalism for expressing *modified* DEMOS models was elaborated. In this chapter the practicality of combining discrete event simulation with a behavioural analysis tool based on a process algebra, generating both from a shared graphical description is demonstrated.

Several tools have appeared which combine simulation and exact *quantitative* solvers using a common input format [106] [8,10]. A series of tools, beginning with the SIMMER Process Interaction Tool [72], have shown the potential for generating DEMOS models from graphical input. The translation of a subset of unmodified DEMOS syntax into workbench code for either CCS or SCCS given by Tofts in [100] was implemented by him as two SML programs. These permit the conversion of DEMOS programs into either process algebra and the use of the Concurrency Workbench to prove properties of the systems. GreatSPN and DSPNExpress, graphically based stochastic Petri net tools, allows both simulation and structural analysis of their underlying place transition net models.

Here, a new tool called Demographer allows both *modified* DEMOS discrete event simulation models and CCS process algebra models to be generated from a common graphical description. The former can be solved by the DEMOS discrete event solver, while the latter can be analysed by the Concurrency Workbench.

5.2 Demographer

Demographer is a simple graphical editor for creating both *modified* DEMOS discrete event simulation models and Calculus of Communicating Systems (CCS) [58] models directly from extended activity diagrams as described in Chapter 4. The current version runs under MS/DOS and is written entirely in SIMULA. An earlier version, using less well defined definitions of extended activity diagrams exists for X Windows under UNIX. Compilation and execution of *modified* DEMOS models is currently done separately, but it is intended that they should be integrated into the graphical front end.

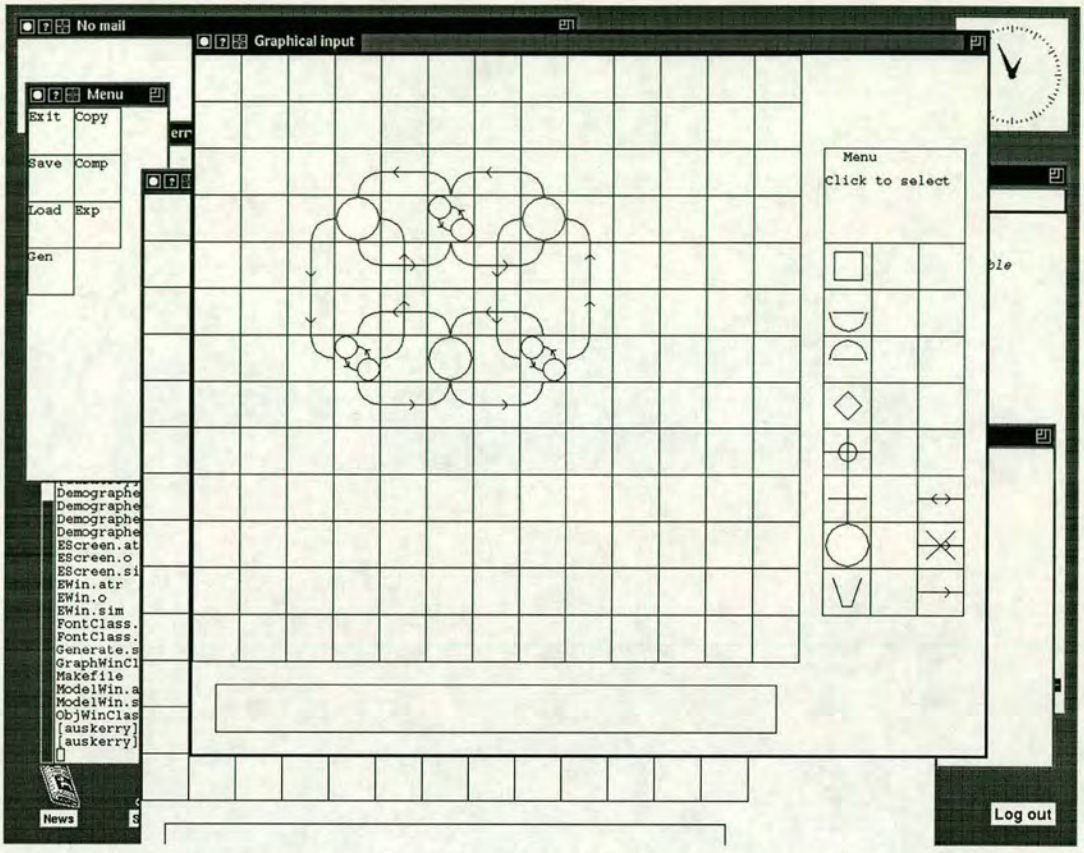
CCS is generated in the syntax of the Concurrency Workbench for most parts of the language. Both the basic calculus and its temporal extension can be generated. The Concurrency Workbench (CWB) remains a separate tool, but it is trivial to load the output of Demographer into it. By integrating the two types of modelling in a pair of compatible tools, the benefits of both approaches are more easily obtained. At the same time the process of modelling is simplified and consistency between the two solvable forms of the model is ensured.

5.2.1 The basic tool

Demographer allows the user to draw enhanced activity diagrams, by selecting symbols from a menu and placing them on a canvas, which is divided into a grid of squares. Each symbol occupies one square in the grid. Symbols are connected by drawing linking symbols in the squares between them. The types of the symbols joined and the direction of the links determine their meaning, in line with the formal grammar for extended activity diagrams developed in Chapter 4.

Many symbols require additional information to be supplied to complete the description of the model. For instance, the *Hold* symbol requires a description of the duration of the delay it represents. Additionally many symbols can usefully be annotated by a short comment or description. This is possible by selecting a symbol and invoking an *open form* operation. This will cause an input form menu appropriate to that symbol to be displayed. The user may then enter the required information by typing into this form.

Figure 5.1: Demographer user interface



When a model's description is believed to be correct and complete the user may request that a DEMOS program be generated from it. This is done by activating the Generate button. The user will then be asked for the name of a file into which the DEMOS source is to be written.

As well as saving the DEMOS source, the user may save and load the graphical representation and annotation. This is stored in a standard format called DIA format, which is common to both versions of Demographer. Thus models created under MS/DOS may be used by the X Windows version by transferring the files, which are in ASCII format. The current MS/DOS version is complete, while the X Windows version may not be able to recognise some symbols. CCS generation is currently a separate program, reading the DIA representation of the model and writing CCS to a new file.

5.3 *modified* DEMOS

Here the redefined version of DEMOS, known as *modified* DEMOS, is outlined. This builds all synchronisation objects, such as resources, with an option for FIFO priority queueing, as in unmodified DEMOS, and an option for releasing blocked Entitys as soon as they are able to proceed. This implements the versions of Acquire, Remove and Take needed to allow equivalence between DEMOS simulation behaviour and that expected by CCS without using FIFO resources etc.

5.3.1 Supporting non-FIFO blocking

The changes to the DEMOS package include a global flag which can be set and reset to force all new synchronisation objects subsequently created to be FIFO or non-FIFO. Within these objects, their behaviour can be modified after creation by calling a setting or a resetting procedure to modify their internal FIFO flag.

5.3.2 Introduction of Store object

The problems with unbounded buffer objects, represented by Bin in unmodified DEMOS, are dealt with by introducing a *Store* object, as defined in Chapter 3. This has two queues, rather like a WaitQ, one for Entitys blocked trying to *Remove* part of the contents of the Store and one for those blocked trying to *Add* to it. As with Res and Bin, Store objects in *modified* DEMOS may use strict FIFO queueing or allow those with smaller requests to proceed if those in front are still blocked.

5.4 Active versus passive objects - a digression

The process view of simulation is built on a distinction between active objects (processes) and passive ones (resources etc.). In Chapter 3 it was necessary to view all objects in the CCS world as active. It is therefore worth considering whether re-implementing DEMOS in this way would lead to any real differences. If so, it would be sensible to do so to ensure consistency with the CCS definition of DEMOS semantics.

To investigate this question, a version of DEMOS was built using only Entity, to model active objects, and CondQ, to model communication. These correspond directly to the CCS primitives of agents and complementary actions. To illustrate the

results of this, the effects on Res are considered. It is typical of the other mechanisms.

5.4.1 Res as an Entity

A *Res* maintains a count of how much of a resource is unused. This amount is set initially and may never exceed its initial value. It supports two interactions with *Entity*s - *Acquire* and *Release*.

Acquire is a request from an *Entity* process for an amount of the resource being modelled. This request blocks the requesting process and can be modelled by its passivation after entering a request queue. In this way it is identical to an *Entity* which enters a DEMOS WAITQ and becomes a slave or which enters a *CondQ* and performs a *WaitUntil* sufficient resource is available. In the former case, the *Res* is very similar to the master process, *Coopt*ing the requesting process by using *Find* to express the condition that its required amount of the resource be less than or equal to that available. In the latter case the *Res* would *Signal* the *CondQ* on receiving a *Release* message from an *Entity*. *Release* increments the amount of resource available and activates the *Res* process either to look for slaves which can now be *Coopt*ed or to *Signal* its *CondQ*.

The choice of which way to represent a *Res* as an *Entity* is therefore unclear. The form which gives the simplest representation and is closest to a CCS model is chosen, i.e. in terms of a *CondQ*.

Figure 5.2: Res as an Entity/CondQ pair - M_Res

```
ENTITY class M_RES(RAmount); integer RAmount;
begin ref(CONDQ) WQ;

   procedure ACQUIRE(Amount); integer Amount;
   begin
      WQ.WaitUntil(Amount<=RAmount and Current==WQ.First);
      RAmount := RAmount - Amount;
   end;

   procedure RELEASE(Amount); integer Amount;
   begin
      RAmount := RAmount + Amount;
      WQ.Signal;
   end;
   WQ :- new CONDQ(Title&"'s Queue");
end;
```

Figure 5.3: Comparison of Res and M_Res traces

Trace using DEMOS RES

```
TIME/ CURRENT      AND ITS ACTION(S)

0.000 DEMOS    SCHEDULES BOAT 1 NOW
               HOLDS FOR 100.00,UNTIL
100.000
      BOAT 1 SCHEDULES BOAT 2 AT 5.000
               SEIZES 2 OF TUGS
               SEIZES 1 OF JETTIES
               HOLDS FOR 3.000, UNTIL
3.000
3.000   RELEASES 2 TO TUGS
               HOLDS FOR 10.000, UNTIL
13.000
5.000 BOAT 2  SCHEDULES BOAT 3 AT 10.000
               SEIZES 2 OF TUGS
               SEIZES 1 OF JETTIES
               HOLDS FOR 3.000, UNTIL
8.000
8.000   RELEASES 2 TO TUGS
               HOLDS FOR 10.000, UNTIL
18.000
10.000 BOAT 3 SCHEDULES BOAT 4 AT 15.000
               SEIZES 2 OF TUGS
               AWAITS 1 OF JETTIES
13.000 BOAT 1 SEIZES 1 OF TUGS
               HOLDS FOR 3.000, UNTIL
16.000
15.000 BOAT 4 SCHEDULES BOAT 5 AT 20.000
               AWAITS 2 OF TUGS
16.000 BOAT 1 RELEASES 1 TO TUGS
               RELEASES 1 TO JETTIES
               ***TERMINATES
      BOAT 3 SEIZES 1 OF JETTIES
               HOLDS FOR 3.000, UNTIL
19.000
18.000 BOAT 2 AWAITS 1 OF TUGS
19.000 BOAT 3 RELEASES 2 TO TUGS
               HOLDS FOR 10.000, UNTIL
29.000
      BOAT 4 SEIZES 2 OF TUGS
               AWAITS 1 OF JETTIES
      BOAT 2 SEIZES 1 OF TUGS
               HOLDS FOR 3.000, UNTIL
22.000
20.000 BOAT 5 SCHEDULES BOAT 6 AT 25.000
               AWAITS 2 OF TUGS
22.000 BOAT 2 RELEASES 1 TO TUGS
               RELEASES 1 TO JETTIES
               ***TERMINATES
      BOAT 4 SEIZES 1 OF JETTIES
               HOLDS FOR 3.000, UNTIL
25.000
```

Trace using M_RES

```
TIME/ CURRENT      AND ITS ACTION(S)

0.000 DEMOS    SCHEDULES TUGS 1 NOW
               SCHEDULES JETTIES 1 NOW
               SCHEDULES BOAT 1 NOW
               HOLDS FOR 100.00,UNTIL
100.000
      TUGS 1 ***TERMINATES
      JETTIES 1     ***TERMINATES
      BOAT 1 SCHEDULES BOAT 2 AT 5.000
               HOLDS FOR 3.000, UNTIL
3.000
3.000   SIGNALS TUGS 1's Que
0              HOLDS FOR 10.000, UNTIL
13.000
5.000 BOAT 2  SCHEDULES BOAT 3 AT 10.000
               HOLDS FOR 3.000, UNTIL
8.000
8.000   SIGNALS TUGS 1's Que
               HOLDS FOR 10.000, UNTIL
18.000
10.000 BOAT 3 SCHEDULES BOAT 4 AT 15.000
               W'UNTIL IN JETTIES 1's
13.000 BOAT 1 HOLDS FOR 3.000, UNTIL
16.000
15.000 BOAT 4 SCHEDULES BOAT 5 AT 20.000
               W'UNTIL IN TUGS 1's Que
16.000 BOAT 1 SIGNALS TUGS 1's Que
               SIGNALS JETTIES 1's
               ***TERMINATES
      BOAT 3 LEAVES JETTIES 1's
               HOLDS FOR 3.000, UNTIL
19.000
18.000 BOAT 2 W'UNTIL IN TUGS 1's Que
19.000 BOAT 3 SIGNALS TUGS 1's Que
               HOLDS FOR 10.000, UNTIL
29.000
      BOAT 4 LEAVES TUGS 1's Que
      W'UNTIL IN JETTIES 1's
      BOAT 2 LEAVES TUGS 1's Que
      HOLDS FOR 3.000, UNTIL 22.000
20.000 BOAT 5SCHEDULES BOAT 6 AT 25.0
      W'UNTIL IN TUGS 1's Que
22.000 BOAT 2 SIGNALS TUGS 1's Que
0              SIGNALS JETTIES 1's
               ***TERMINATES
      BOAT 4 LEAVES JETTIES 1's
 HOLDS FOR 3.000, UNTIL 25.000
```

5.4.2 Testing M_Res

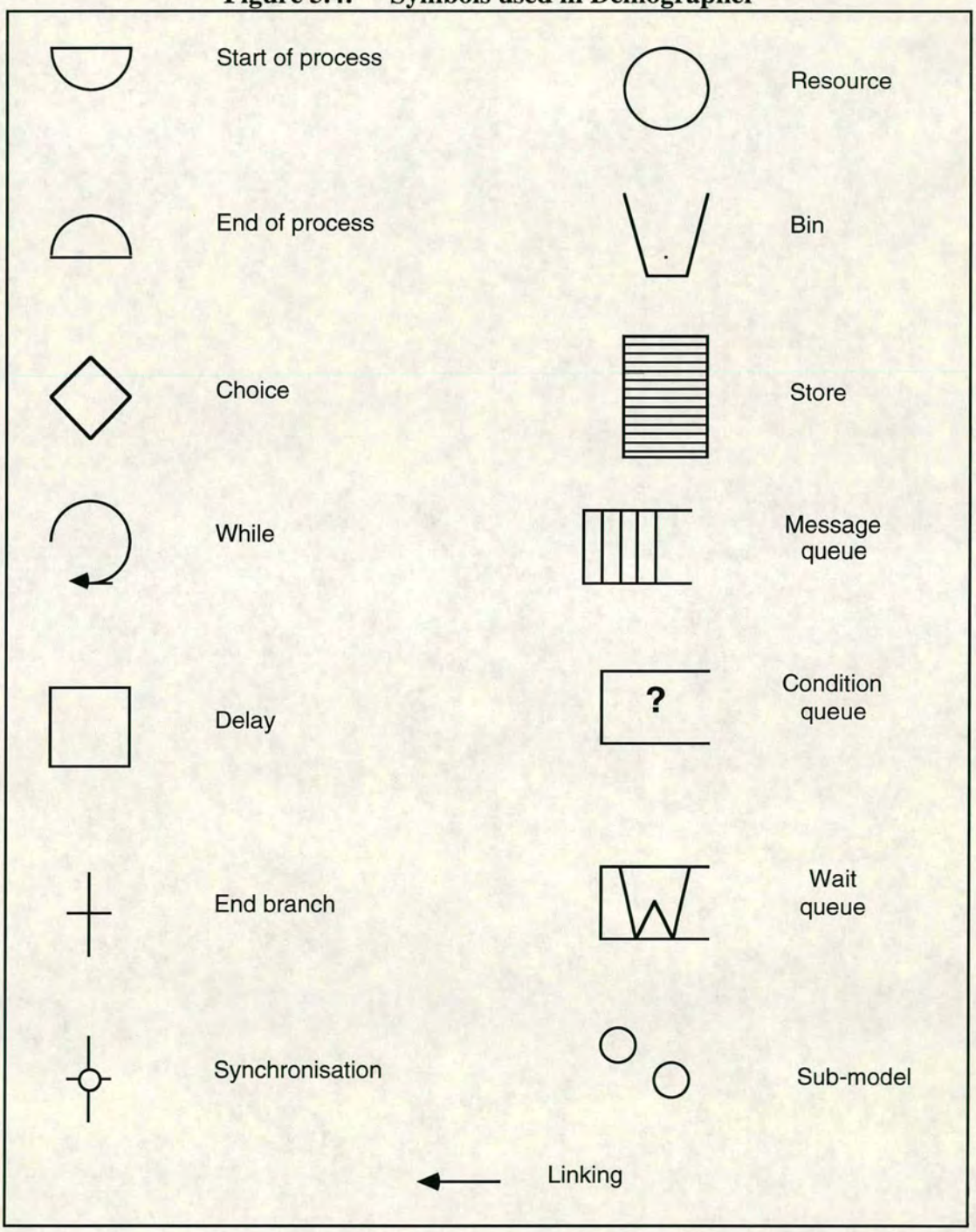
To verify the behaviour of *M_Res*, the harbour model from Birtwistle was modified to use *M_Res* rather than *Res* and the traces compared. These are given above. A

similar approach was used to verify the other re-writings. Despite certain differences in the trace messages, the sequence of actions is identical.

5.5 The current set of symbols

The set of symbols described here is essentially the same as that described in Chapter 4. Their connection into a graph has to match the grammar in Figure 4.16.

Figure 5.4: Symbols used in Demographer



5.5.1 Linking

The extended activity diagram grammar specifies only a general topological notion of placing and linking of symbols. The precise mapping onto a display medium is left to the implementor. In particular, linking is likely to depend on the format of the canvas used. In Demographer a grid of squares is assumed, with one node per non-empty square. Links are composed of directed link nodes, joining the objects nodes, which is rather restrictive in terms of the number of paths possible between exit side and entrance side of the nodes being joined, but seems adequate in most cases. This works for the linking definitions in the grammar, since no node is required to have more than one link attached to any side. As well as upwards, downwards, leftwards and rightwards, link nodes can indicate changing direction by ninety degrees and crossing of links.

Placing two flow of control nodes in adjacent squares is interpreted as linking them if that makes sense in terms of the grammar.

A sub-model node may have several links to it. This is supported in the grammar by requiring synchronisation and hold nodes to be attached beneath the sub-model so that links may be made to them. The synchronisation nodes which match parameters must be attached to objects in a top to bottom order which matches the order of the parameters of that sub-model (see below).

5.6 Attributes of symbols

As was remarked at the end of Chapter 4, by suitable definition of attributes for the nodes in an extended activity diagram, a complete model or sub-model can be generated automatically, either in DEMOS or in (T)CCS. This section describes those attributes required in Demographer at present. Most are concerned with DEMOS code generation. Demographer allows up to six attributes per node, presenting a menu with a line, starting with a prompt, for text to be entered for each attribute that is required for that node type.

5.6.1 Attribute grammars for activity diagrams

A similar approach was used successfully in the earliest versions of the Process Interaction Tool [72]. In this and subsequent PIT tools [6] an elaborate language for the definition of annotated graphs was developed, known as Graph Definition

Language (GDL). GDL led to a generic graph editing tool [70,72], which was customised by reading in a GDL file when starting up. This file defined node types, their appearance, how they could be linked and what attributes they should have in their form menus. It was necessary, however, to write from scratch a backend processor for the data structures produced for any given graph type. In later versions, GDL was extended to be the language in which models were stored as well.

Having produced in this dissertation a formal grammar for extended activity diagrams, it is now clear that GDL was acting as an attribute grammar meta-language. Unfortunately this insight was not available at that time and it is very clumsy when viewed in this light. Perhaps as a result of this, the true power of attribute grammars, as used in compiler compilers, was not exploited, namely the ability automatically to generate required output from the attribute definitions. Although the MS/DOS version of Demographer uses no GDL form of input to drive it, the partial X Windows version has shown that this is possible for at least a subset of the symbols. No further claims are made at the present, but it seems likely that such an approach will lead to a truly general graphical editing and synthesising tool

5.6.2 Attributes and properties of symbols in Demographer

Although up to six attributes are allowed, most symbols use fewer. In the following list, those with an asterisk are optional, i.e. may be left blank and still allow a model to be generated. As well as the attributes, some idea of the corresponding DEMOS code is given. The CCS follows the correspondences to DEMOS worked out in Chapter 3, as far as could be achieved before work halted.

5.6.3 Flow of control symbols

Start symbol

A process starts with a start symbol. If several start symbols are drawn above each other, a corresponding number of instances of that type of process is to be generated.

The start symbol requires the user to specify:

a name, used to define the entity class and a reference to an instance of it. The class name has the suffix "_C" appended. Currently each instance of the same process has a separate (identical) class definition.

an initial scheduling delay used literally as given as a parameter to Schedule for the instance after its creation.

an inter-arrival time * if empty, this field is ignored, if used, the type of a distribution (such as NegExp) and its parameters, as required by DEMOS, should be given. A suitable unique distribution is generated, whose name is a combination of the *name* in this start node and a suffix meaning arrivals. The first action of the process will now be to schedule its successor according to this distribution, generating a stream of arrivals.

three lines for declaration of *local variables*. These will be inserted exactly as typed at the start of the class for this process. Although intended to support conditional expressions, any legal SIMULA declarations or statements are accepted here.

End symbol

This indicates the end of a sequence of symbols intended to represent one process. No annotation is required.

Hold symbol

This indicates a delay for some activity.

It requires

a reason which will be enclosed in comment delimiters ('!' and ';') and inserted into the process actions before the Hold.

a delay currently taken literally as the text of the parameter to the corresponding Hold statement in the DEMOS program.

Choice symbol

This allows the model to choose to follow one of two paths until some future joining, which is marked by an end-branch symbol. The second of these branches may be omitted. This corresponds to the if-then-else and the simple if-then in a programming language.

It requires

a reason which is inserted into a comment in the same manner as for *hold.*

a condition which will be used to generate a value of True or False. Currently this is inserted exactly as typed between the words `if` and `then` in the DEMOS program.

The path followed when the condition is True is that leading from the bottom of the choice symbol. When the condition is false, the path to the left or right is followed. Testing for the existence of the else branch is done in that order. If no path in either direction is found, the program simply skips the true branch if the condition is False and carries on from the end branch symbol.

While symbol

This is rather like the choice symbol and has the same annotations.

Instead of performing the true branch once only, it continues to repeat it as long as the condition remains True. If condition is given as the literal "true" or is never altered, the loop will continue indefinitely. There is no else branch to a while and none is checked for. Again the extent of the loop is marked by an end-branch symbol.

End-branch symbol

This marks the end of a sub-part of the process' behaviour, currently the branches of a condition or the body of a while loop. It has no attributes.

Synchronisation symbol

This indicates that an action defined by a link to or from an external synchronisation mechanism is to take place before continuing. Currently an incoming (outgoing) link has the following meanings for synchronisationnodes within process graphs:

External node type	Incoming link	Outgoing link	Parameter required
Resource	Acquire	Release	Amount - integer
Bin	Take	Give	Amount - integer
Store	Remove	Add	Amount - integer
Message Queue	Receive	Send	Object - ref (Message)
Condition Queue	Wait Until	Signal	Condition - Boolean by name
Wait Queue	Coopt	Wait	None
Synchronisation Node		Schedule	Delay - real
Start or Submodel Node		Schedule	Delay - real
Hold		Interrupt	Signal - integer

It requires

a reason which is used as a comment.

a parameter value which will be of the type shown in column 3 above.

Synchronisation nodes are also used below submodel nodes, to define parameters to and scheduling of submodel entities. Again the meaning of a link is determined by the type of node to which a link is made. There is no code generated for the scheduling and interrupting links, as this has already been generated within the submodel's code. The outgoing links all bind actual objects to the formal parameters of the submodel. Incoming links are only significant for the node on the other end, which cannot be a passive object.

External node type	Incoming link	Outgoing link	Parameter required
Resource	Not used.	ref(Res) param	Not used
Bin	No used	ref(Bin) param	Not used
Store	Not used	ref(Store) param	Not used
Message Queue	Not used	ref(MessageQ) param	Not used
Condition Queue	Not used	ref(CondQ) param	Not used
Wait Queue	Not used	ref(WaitQ) param	Not used
Synchronisation Node		ref(Entity) param	Not used
Start Node		ref(Entity) param	Not used
Hold		ref(Entity) param	Not used

5.6.4 Passive object symbols

The following symbols represent objects outside process descriptions, passive objects. The meaning of a link is fixed by the type of node to which a link is made.

Resource

This corresponds to a DEMOS Res. It requires:

a name - used to build a ref (Res) declaration, a new Res statement and to tag any Acquire and Release calls in processes linked to this *resource*.

an amount - used in new Res statement as initial amount & limit of res.

Bin

This corresponds to a DEMOS Bin. It requires:

a name - used to build a ref (Bin) declaration, a new Bin statement and to tag any Take and Give calls in processes linked to this *bin*.

an amount - used in the new Bin statement as the initial amount held.

Store

This corresponds to a DEMOS Store. It requires:

a name - used to build a ref (Store) declaration, a new Store statement and to tag any Remove and Add calls in processes linked to this *store*.

an amount - used in the new Store statement as the initial amount held.

a limit - used in the new Store statement as the limit of capacity.

Message Queue

This corresponds to a DEMOS MessageQ. It requires:

a name - used to build a ref (MessageQ) declaration, a new MessageQstatement and to tag any Send and Receive calls in processes linked to this *message queue*.

Condition Queue

This corresponds to a DEMOS CondQ. It requires:

a name - used to build a ref (CondQ) declaration, a new CondQ statement and to tag any Signal and WaitUntil calls in processes linked to this *condition queue*.

a Boolean flag all - used to control the extent of searching when a Signal is received.

Wait Queue

This corresponds to a DEMOS WaitQ. It requires:

a name - used to build a ref (WaitQ) declaration, a new WaitQ statement and to tag any Wait and Coopt calls in processes linked to this *wait queue*.

Sub-model

This corresponds to a separately defined process. See section 5.7.4 for details. It requires:

a name - used to build a ref (Name) declaration, a new Name statement and to tag any schedules or interrupts to this submodel.
A parameter name and type list - used to identify the types of objects to which this node's dependent synchronisation nodes should be attached.

5.7 Implementation

The general operation of Demographer is described in this section. Although much is independent of a particular version, some aspects refer to either the MS/DOS or the X Windows version. Various formats and representations are used at different stages, stored in ASCII files. Figures 5.5 and 5.6 show the structure of the two current versions.

Figure 5.5: Structure of files in the MS/DOS version of Demographer

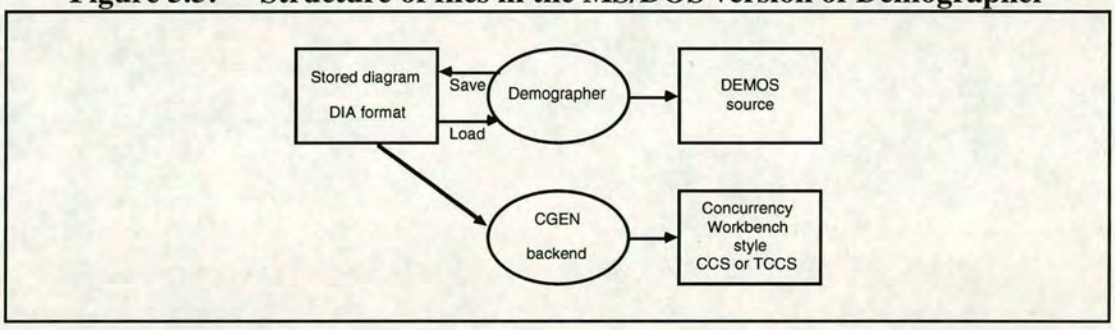
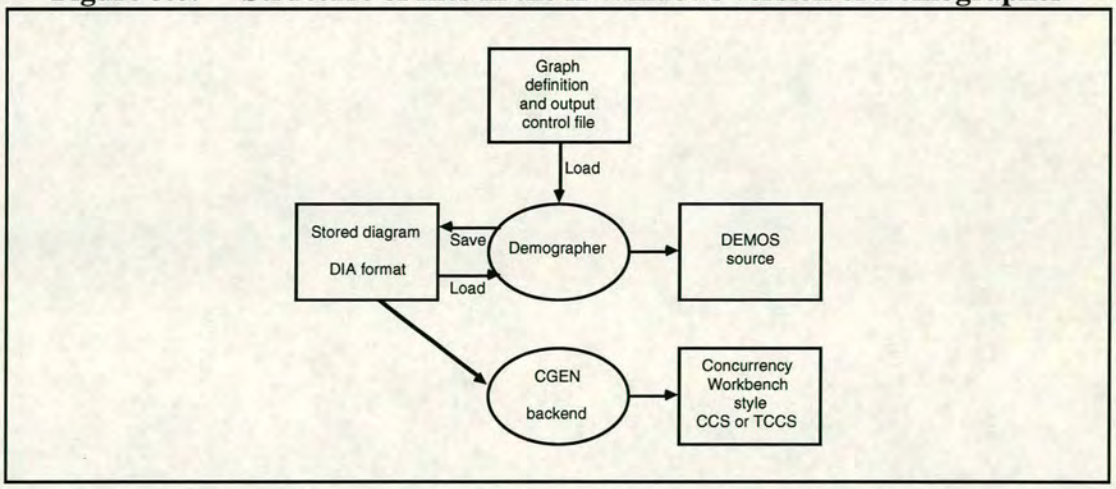


Figure 5.6: Structure of files in the X Windows version of Demographer



5.7.1 Loading and saving models

Demographer begins by asking for the name of an input file. If given, this should be in the DIA format used by all versions of Demographer. If no name is given, an empty canvas is created. If a valid file name is given, the canvas will appear with the corresponding diagram displayed and the underlying annotations will also have been loaded.

The format of stored diagrams is a simple minded representation of the grid, its nodes and their attributes. For each non-empty square in the grid the following is output in a fixed format:

> X and Y co-ordinates in the grid,
>
> type of node as an integer,
>
> the text entered into each of the six possible fields holding attributes.

Although this is not very compact, it is simple and complete. Links are represented as chains of appropriately directed and overlaid directed link nodes for the squares they cross. The essence of the DIA representation is that the complete description of the grid is enough to define the model.

5.7.2 Interpreting the diagram

Demographer works at three levels when interpreting an activity diagram.

> It begins by making a complete scan of the grid, locating all nodes corresponding to *objects* in the grammar. A linked list of records is created for these.
>
> When generating output, *processes* are parsed in a simple recursive descent manner, by following flow of control links from each start node and *sub-models* are parsed by interpreting the chain of synchronisation nodes linked beneath them.

When a *synchronisation node* is found during the parsing of a process or sub-model, its type and partner are established by following, forwards or backwards, the link to the other node in its node-link-node triple. Where a synchronisation node has links attached to both sides, this is treated as two synchronisation nodes, the first with the left link , the second with the right link attached.

5.7.3 Generating flat models

The generation of models is done in three passes through the data gained by scanning the grid.

First the start nodes on the object list are used to identify the processes described and to output an Entity, in DEMOS, or a binding of an agent to an identifier, in CCS, for each of these. In model generation, the mapping of processes is quite simple, the only outside information being found in objects at the other end of links to or from synchronisation nodes. Any inter-arrival time distributions found in the start node are added to the object list at this time.

A second pass down the object list is then used to generate declarations of ref variables for all objects in a DEMOS model. This is again straightforward, involving the use of the *name* field to create an identifier of appropriate type. For an entity the type will be the *name* with the suffix "_c", as for the identifier in the corresponding Entity declaration. This pass produces no output in a CCS model.

The third pass generates instances of objects. In DEMOS this means *new* statements, with identifier, class name and title all generated from the *name* field. Other parameters to the objects are found from other attribute fields as defined above. In CCS, instances are bound by parallel composition with a DEMOS agent and appropriate restriction of label visibility. Passive objects are generated according to their templates defined in Chapter 3, in some cases, such as resources, having their extent defined by a parameter attribute.

5.7.4 Generating hierarchical models

There are two stages in the construction of a hierarchical model using Demographer, building of components and assembly of models. The first of these is further sub-divided into building of atomic processes and assembly of compound processes. At

the moment they are assumed to work bottom up, but this constraint should be relaxed in future versions.

Building an atomic component process

In Chapter 3 the general notion of component based process oriented modelling was introduced. Demographer follows this view quite closely. Thus the initial task is to construct the lowest level, atomic processes to be used. Since these are intended to be re-usable, they must retain all information necessary for their incorporation in higher level, compound components or complete models. An atomic process description in Demographer consists of a single process described by a start/end node pair and their linking flow of control nodes. All communications and interactions with other objects are shown by including those objects. In the case of a Schedule call to another process a submodel node is used, with no dependent synchronisation nodes. No attributes, other than their Name fields are used in these object nodes. These names are used as the identifiers of the formal parameters of the Entity sub-class generated.

Generation of DEMOS code proceeds in the same way as for a complete model, except that the ref variable declarations in the full model are replaced with the building of a formal parameter list in the header of the Entity sub-class. A list of the identifiers and types of all parameters id also generated, automatically, in the Params field of the form of the Start node. This will be used when importing the sub-model at higher levels.

CCS generation is fairly straightforward for atomic processes, except for nested structures, i.e. loops, where dummy sub-agent names are created to help with recursion, and conditions, where only a representative sub-range of possibilities are tested, for simple comparisons or a place marker is generated for anything more complicated. Again, where Bins and certain queue structures are required, only a subset of possible values are generated.

Assembly of compound processes

A compound process description can contain all the elements of a full model. This means that some means of distinguishing objects local to the generated compound process (those inside the shaded areas in Chapter 4) and objects to be left external. In the generated DEMOS, those locally defined will have to be treated in the same way

as processes in a full model, with ref declarations and new statements being included in the body of the Entity sub-class, while those left external will be added to the formal parameter list. One and only one full process description is required and allowed in a compound process description. This will control the scheduling by the generated Entity body of the locally defined sub-processes.

Sub-model nodes are used to introduced predefined sub-processes. These require a name, which is used to locate files describing their external interface. Once this is supplied, they can construct a parameter identifier/type list, from information output when their underlying code was generated. It is currently left to the user to add a corresponding number of synchronisation and hold nodes below the sub-model. These must be linked in the order of the parameter list to the objects intended as the actual parameters. This may sound cumbersome, but works reasonably effortlessly. Future version of the tool will generate the synchronisation nodes automatically.

Thus DEMOS code generation proceeds as a combination of full model and atomic process code generation, with any unsatisfied parameters (unmatched links) being propagated out by adding them to this Entity's formal parameter list. Satisfied parameters are supplied with appropriate ref variable identifiers in the actual parameter list within Entity new statements. Unsatisfied parameters get the formal parameter name used to pass them out.

CCS generation is more complicated for this level, since parameter matching in DEMOS corresponds to renaming and hiding in CCS. Thus an analogous phase of link matching is performed.

Hierarchical model assembly

At the top level model assembly proceeds as a combination of flat model generation and compound process generation. No outward parameter propagation is possible at this level, of course. No detailed process description is needed for the main program. Any process descriptions at this level are treated as new Entity definitions.

5.8 Conclusions and further work

The current version of Demographer is as complete as was needed to produce this dissertation. It demonstrates that all the features defined in the graphical language of extended activity diagrams can be automatically converted into DEMOS and that

many of them can be automatically converted into CCS. The limitations on the latter stem from the lack of support in either CCS or TCCS for stochastic and continuous values. This makes it impossible to deal with time in the manner required in quantitative modelling. It can be argued that DEMOS only supports the usual digital computer's discrete approximation to continuous values, but this can only be modelled realistically in CCS or TCCS for very restricted ranges of value, with any kind of accuracy.

The problems of stochastic variables is less important, as noted elsewhere, since a range of important properties can be shown to hold for any branching probabilities or rates. Such models are conservative in what they predict as safe, but are often still of use. More hopefully, a number of new probabilistic and stochastic process algebras, such as TIPP [31] and PEPA [37,30], are emerging which include the desired features. It is an important continuation of this work to investigate the use of mappings from extended activity diagrams into these algebras.

Chapter 6

Exploiting CCS for Simulation Models

6.1 What modellers need to know

The use of functional properties is fuelled by a number of questions in the minds of modellers. In this chapter some of the most important are examined in the context of CCS and the modal μ-calculus as a means for reasoning about them. Among the questions that might be tackled are the following.

1 **Does a simplification change behaviour?**

In order to make execution of simulation models tractable, it is often desirable to simplify areas of detail. This leads to questions such as: "Can the detailed modelling of this sub-model be replaced by a stochastically determined hold?" and "Does it change the behaviour of the model if I replace a sub-model with a formula?"

There are two approaches to dealing with this sort of question. The first is to formulate rules for simplifications which are guaranteed to leave behaviour unaltered. Since models are expected, at least in part, to be generated by composing predefined instances of components, it would be unsurprising if this did not cause redundant states to be included. The second approach is to make some simplification and to have a means of testing whether important properties are unaltered.

2 **Does the model's implementation mask a problem?**

Since discrete event models are actually executed in an interleaved manner, rather than in an asynchronously concurrent one, it is difficult to guarantee that

the modeller's intentions are reflected by the behaviour of the system. Examples include conditional waiting intended to model genuinely concurrent enabling of blocked processes, as in the CSMA/CD protocol of Ethernet, and implementation of acquire which hides starvation, as in the standard DEMOS version of reader/writer locking.

3 **Are there implications of structuring the model hierarchically?**

For ease of expression and reuse of sub-models modellers may need to know if a hierarchical model corresponds to its flat equivalent or behaves as expected overall. This requires ways of testing that important properties are or remain true in a hierarchical model.

Having identified the sorts of questions that modellers might want to ask, it is now possible to examine how successfully they are addressed by testing their CCS equivalents. In the rest of this chapter the problems identified above are considered in turn by the use of typical examples. It is clear that CCS offers considerable potential, but it is not clear yet where its limits lie.

6.2 Simplification of models

In this section the two approaches to simplification described above are considered in turn. First the possibility of identifying, from the CCS model, simplifications which leave the simulation model's behaviour unaltered is considered. Then ways of identifying equivalence of models simplified by intuition are considered.

6.2.1 Identification of redundancy in models

Some conditions for eliminating actions and states are identifiable in terms of the CCS representation of a model. In some cases these may in themselves identify useful information about the system being modelled.

Elimination of transitions

To show how elimination of potential transitions is possible, irrespective of timings, the example in figure 6.1 uses the basic Calculus, as it would be generated automatically by Demographer, to model a simpler version of the harbour model from section 3.5.1, creating only three, terminating boats and reducing the initial number of tugs to two. This simplifies any analysis, and it will be used again to investigate deadlock.

Figure 6.1: CCS of a simple harbour model

$BOAT$	$\stackrel{\text{def}}{=}$	$\overline{jAcq_1} \,.\, \overline{tugAcq_2} \,.\, \overline{tugRel_2} \,.\, \overline{tugAcq_1} \,.\, \overline{tugRel_1} \,.\, \overline{jRel_1}$
$TUGS_2$	$\stackrel{\text{def}}{=}$	$(tugAcq_1.TUGS_1) + (tugAcq_2.TUGS_0)$
$TUGS_1$	$\stackrel{\text{def}}{=}$	$(tugAcq_1.TUGS_0) + (tugRel_1.TUGS_2)$
$TUGS_0$	$\stackrel{\text{def}}{=}$	$(tugRel_1.TUGS_1) + (tugRel_2.TUGS_2)$
$JETTIES_2$	$\stackrel{\text{def}}{=}$	$(jAcq_1.JETTIES_1) + (jAcq_2.JETTIES_0)$
$JETTIES_1$	$\stackrel{\text{def}}{=}$	$(jAcq_1.JETTIES_0) + (jRel_1.JETTIES_2)$
$JETTIES_0$	$\stackrel{\text{def}}{=}$	$(jRel_1.JETTIES_1) + (jRel_2.JETTIES_2)$
$MODEL$	$\stackrel{\text{def}}{=}$	$(TUGS_2 \mid JETTIES_2 \mid BOAT \mid BOAT \mid BOAT) \setminus \mathfrak{L}(MODEL)$

Restricting *MODEL* by the sort of its unrestricted self, $\mathfrak{L}$, allows its components to be simplified, since they can no longer engage in any outside communications, only in τs. By applying the results of the CCS Expansion Law [58], any choices in agents within *MODEL* which begin with a label in $\mathfrak{M}$, where $\mathfrak{M}$ contains those labels not matched by a complementary action within the same scope, i.e. which are not partners in τs, may be eliminated. In other words, if the model is prevented from engaging in any outside activity, any branches guarded by actions which cannot be satisfied internally can be pruned without changing the overall behaviour of the model.

This significantly reduces the complexity of the Jetties resource, as shown in Figure 6.2, allowing edges in the resulting transition graph to be removed. In the current example it does not lead to a simpler simulation model, but in some models it would have an even greater effect, eliminating states in the transition graph not just transitions. State elimination is examined in the next section.

Figure 6.2: Simplified CCS of Jetties resource from Harbour model

$JETTIES_2$	$\stackrel{\text{def}}{=}$	$(jAcq_1.JETTIES_1)$
$JETTIES_1$	$\stackrel{\text{def}}{=}$	$(jAcq_1.JETTIES_0) + (jRel_1.JETTIES_2)$
$JETTIES_0$	$\stackrel{\text{def}}{=}$	$(jRel_1.JETTIES_1)$

In fact such simplifications are possible at any point where restriction is applied. This means that each sub-model definition is a potential point for elimination of transitions. Taking the reader/writer model from section 4.4.1, as it would appear if the two processes were first modelled separately and then combined, the resulting CCS model is as given in Figure 6.3.

Figure 6.3: Hierarchically constructed Reader/Writer model in CCS

$$
\begin{array}{lcl}
Reader & \stackrel{def}{=} & \overline{buffRAcq_1} \,.\, \overline{buffRRel_1} \,.Reader \\
Writer & \stackrel{def}{=} & \overline{buffWAcq_3} \,.\, \overline{buffWRel_3} \,.Writer \\
SharedBuff_3 & \stackrel{def}{=} & buffSAcq_1.SharedBuff_2 + buffSAcq_2.SharedBuff_1 + buffSAcq_3.SharedBuff_0 \\
SharedBuff_2 & \stackrel{def}{=} & buffSAcq_1.SharedBuff_1 + buffSAcq_2.SharedBuff_0 + buffSRel_1.SharedBuff_3 \\
SharedBuff_1 & \stackrel{def}{=} & buffSAcq_1.SharedBuff_0 + buffSRel_2.SharedBuff_3 + buffSRel_1.SharedBuff_2 \\
SharedBuff_0 & \stackrel{def}{=} & buffSRel_3.SharedBuff_3 + buffSRel_2.SharedBuff_2 + buffSRel_1.SharedBuff \\
Model & \stackrel{def}{=} & \left(\begin{array}{l} Reader\left[buffSAcq_1/buffRAcq_1, buffSRel_1/buffRRel_1\right] \mid \\ Writer\left[buffSAcq_3/buffWAcq_3, buffSRel_3/buffWRel_3\right] \mid \\ Writer\left[buffSAcq_3/buffWAcq_3, buffSRel_3/buffWRel_3\right] \mid \\ SharedBuff_3 \end{array} \right) \backslash \left\{ \begin{array}{l} buffSAcq_1, buffSAcq_2, \\ buffSAcq_3, buffSRel_1, \\ buffSRel_2, buffSRel_3 \end{array} \right\}
\end{array}
$$

As in the harbour model, not all of the possible actions in the resource are matched by complementary ones in Reader or Writer. They cannot be removed until they are restricted, but can then form a simplification. The Buffers resource no longer needs those actions using $buffSAcq_2$ or $buffSRel_2$ and these are eliminated below.

Figure 6.4: Reduced form of Buffers resource

$$
\begin{array}{lcllcl}
SharedBuff_3 & \stackrel{def}{=} & buffSAcq_1.SharedBuff_2 & + & buffSAcq_3.SharedBuff_0 \\
SharedBuff_2 & \stackrel{def}{=} & buffSAcq_1.SharedBuff_1 & + & buffSRel_1.SharedBuff_3 \\
SharedBuff_1 & \stackrel{def}{=} & buffSAcq_1.SharedBuff_0 & + & buffSRel_1.SharedBuff_2 \\
SharedBuff_0 & \stackrel{def}{=} & buffSRel_3.SharedBuff_3 & + & buffSRel_1.SharedBuff_1
\end{array}
$$

Eliminating complete states

One major claim for Petri net models as a formalism for simulations has been their ability to identify redundant states, in the context of a particular marking, and so

allow simplification of the model at run time. Yücesan and Schruben also show how to eliminate states in their event based simulation formalism. In very simple models, such as the previous two examples, CCS can help to reduce the number of paths between states in a model. This in turn can simplify the analysis of the behaviour of the model. The question remains as to whether it is ever possible in the CCS model to eliminate states completely.

In terms of a state transition diagram, elimination of a potential state in a sub-model is possible if there are no edges entering it in the combined model where it is used. In CCS terms, this means that no agent corresponding to a certain state in the sub-model agent is ever activated as the result of an action in the overall model agent. In terms of the reader/writer example, this could mean that, for instance, the agent $SharedBuff_2$ could be shown never to follow any of the actions possible in *Model.* By re-formulating that model with both readers and writers working in units of two buffers, such a condition is easily created.

Following the reasoning above, a simple example of a model where certain levels of resource are unreachable is now created. In such a simple case this may seem trivial, but in more complex models, such possibilities may be far from obvious. Consider a simple factory model, where there are two machines, a *Mill* and a *Polisher.* The *Mill* shapes pieces and then passes them to the *Polisher.* Since the pieces are long, the *Mill* cannot begin work on a new piece until the *Polisher* has half finished its current piece. Pieces are transferred on dollies. The *Mill* loads its ingots from two *Dollies*, while four are needed by the *Polisher* to move milled pieces for polishing. At the halfway point the *Polisher* can release two *Dollies.*

Figure 6.5: A simple model for unused resource states

Mill	$\stackrel{\text{def}}{=}$	$\overline{dollyAcq_2}\,.\,\overline{dollyRel_2}\,.halfDone.Mill$
Polisher	$\stackrel{\text{def}}{=}$	$\overline{dollyAcq_4}\,.\,\overline{dollyRel_2}\,.\,\overline{halfDone}\,.\,\overline{dollyRel_2}\,.Polisher$

$$Dollies_5 \stackrel{def}{=} \sum_{i=1}^{5} dollyAcq_i.Dollies_{5-i}$$

$$Dollies_n \stackrel{def}{=} \sum_{i=1}^{n} dollyAcq_i.Dollies_{n-i} + \sum_{i=1}^{5-n} dollyRel_i.Dollies_{n+i},$$

where $0<n<5$

$$Dollies_0 \stackrel{def}{=} \sum_{i=1}^{n} dollyRel_i.Dollies_i$$

$$Factory \stackrel{def}{=} (Mill \mid Polisher \mid Dollies_5) \backslash L(Factory)$$

Intuitively it seems that the number of *Dollies* can never reach an even number. This can be verified easily in this simple example. This fact, together with the earlier rule, reduces the resource $Dollies_5$ as shown in Figure 6.6.

Figure 6.6: Dollies resource with redundant states eliminated

$$Dollies_5 \stackrel{def}{=} dollyAcq_2.Dollies_3 + dollyAcq_4.Dollies_1$$

$$Dollies_3 \stackrel{def}{=} dollyRel_2.Dollies_5$$

$$Dollies_1 \stackrel{def}{=} dollyRel_2.Dollies_3$$

This is clearly simpler to reason about, although its usefulness in simplifying a simulation may be small unless it is being very carefully coded. Since it is hoped that resources will be built-in primitives in any simulation package, it would be necessary to have ways of recognising useful simplifications. One such case would be where the upper limit of the amount of resource in use was unreachable. This is clearly true in the example, as no state with the resource, $Dollies_0$, is ever reached. Thus the resource can be further modified as shown in Figure 6.7.

Figure 6.7: Dollies resource normalised to zero lower bound

$$Dollies_4 \stackrel{def}{=} dollyAcq_2.Dollies_2 + dollyAcq_4.Dollies_0$$

$$Dollies_2 \stackrel{def}{=} dollyRel_2.Dollies_4$$

$$Dollies_0 \stackrel{def}{=} dollyRel_2.Dollies_2 + dollyRel_4.Dollies_4$$

The failure of the original resource to reach zero implies that there is more resource available than the model can make use of. This can be used to establish a tuning of the system being modelled and can also allow the simulation to normalise the amount of resource at a lower level. For full generality, it should be noted that:

> there is a surplus n of resource *Res* when for all i in the range 0..n-1, Res_i is unreachable.

In the case given, a further simplification can be made, since those resource states involving odd numbers of dollies are unreachable. This permits the use of a resource with a unit representing two dollies, which can be reflected in the DEMOS model as well as reducing complexity in reasoning about the model. An alternative use of the same kind of analysis is found where the amount of a Bin or a Store can be shown never to exceed a certain limit. This is useful in providing a bound for the Bin (making it into a Store) or reducing the bound on a Store.

Unreachable states in processes

The examples so far have shown simplification of passive objects, such as resources. It is also interesting to investigate state elimination in processes. The simplest case is again the elimination of alternatives in choices which are prefixed by unmatched actions. This is quite likely to happen in reusing components within models. A more complex situation occurs where the actions appear to be matched, but the unwinding of earlier actions absorbs their complement. Put simply, an action and its complement must appear together in at least one state for an agent prefixed by that action to be reachable. The simple example in Figure 6.8 shows both of these in terms of an office services bureau and a messenger service, where the messenger service only accepts one delivery per day.

Figure 6.8: Processes with redundant states in CCS

$$Bureau \stackrel{def}{=} \overline{typing}.C_1 + \overline{copying}.C_2 + \overline{printing}.C_3$$

$$Messenger \stackrel{def}{=} typing.D_1 + copying.D_2$$

$$Model \stackrel{def}{=} (Messenger \mid Bureau)\backslash\{typing, copying, printing\}$$

In the agent *Model*, the *Bureau* cannot evolve into C_3, but can evolve into C_1 or C_2, because the *Messenger* service will not accept printing from it. It is important to note

that this occurs as far as CCS is concerned because of the restriction of the label *printing*, which prevents any outside agent combining further to provide the complement to $\overline{printing}$, which represents the absence of an alternative transport service. This is a simple case. Now consider the agent in Figure 6.9. In *Problem*, *Emergency* cannot become C_3 or C_1.

Figure 6.9: Pre-emptive action removing successor states

$$Emergency \stackrel{def}{=} \overline{typing}.Bureau$$

$$Problem \stackrel{def}{=} (Emergency \mid Messenger)\backslash\{typing,copying,printing\}$$

To test for such cases, the modal μ-calculus at first seems likely to be useful. It can provide the answer to the question, "In all successor states is there any where further progress is impossible?" and thus one possibility of eliminating redundant paths is established. This can be written, using the Box operator defined in Chapter 2, as:

```
Box <->T
```

which is the same as establishing deadlock freedom. It asks whether a dead-end can be reached, but not how many dead ends there are nor which states they are. If such a dead-end does exist and can be located it may reveal unnecessary dead states which were included on the assumption that deadlock could not happen. It does reveal that the model is not well behaved in reaching a steady state, since a dead-end is an absorbing state, killing the model. In general this is a sign of an error in the model's formulation. It is shown in section 6.3.3 below that the Concurrency Workbench provides a way of testing for deadlock which allows all states to be fully identified. Unfortunately it does not provide an answer to the original question. In fact there is no obvious way of exploiting a transition or reachability graph view, which starts from an initial state and generates successor states, to find *unreachable* states in some general graph of states.

A more useful question is to establish whether any sub-states in the component processes do not lie on the paths reachable from the start state of the combined model. At first sight this seems impossible to answer for the general case. Special conditions were identified above which may be used when dealing with resources and buffers to establish whether they can be simplified. A similar method, removing

agents from the definition of components and establishing that combined behaviour is unchanged, can be applied more generally, but is rather a brute force approach without some insight into likely cases. If such insight is available, the approach of section 6.2.2 is applicable.

Returning to built-in Concurrency Workbench functions, the `min` command binds to an identifier the smallest model which is observationally equivalent to a given model. This is an interesting possibility for simplifications, but may lead to re-formulations which no longer have recognisable DEMOS equivalents. Also, the reduced model is defined by the Workbench in terms of meaningless names and there seems no easy way of relating these back to the original.

The Workbench also allows the complete set of reachable agents (states) to be generated for an agent (model). This offers another approach. The existence of unreachable agents results from the restriction of certain labels when the model is composed. Otherwise the workbench assumes that externally generated actions are always possible. By generating the reachable states both with and without such restriction, comparisons may be made to establish which states are redundant. This is shown to be successful in the testing of these examples in Appendix C.

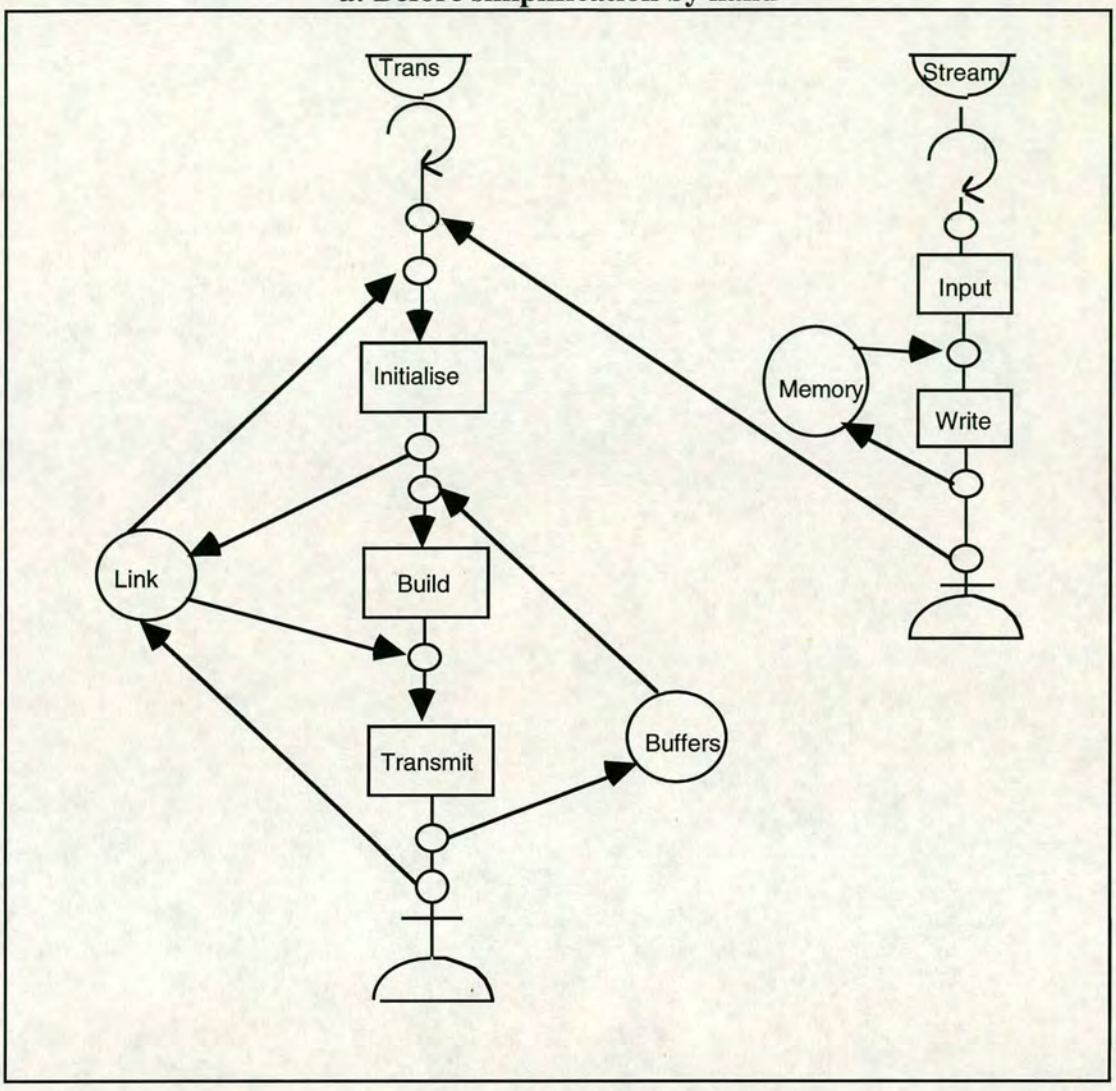
6.2.2 Comparing models simplified by hand

The danger with such sophisticated approaches to understanding behaviour is that they will not be attempted by those who see themselves primarily as *simulation modellers* and not as concurrency experts or formal modellers. In most cases the simplification is carried out in an informal manner and modellers are unable to establish whether such modifications are dangerous, except by running the resulting simulation models and examining their traces. It is therefore worth considering how far it is possible for a simulation modeller to ask whether an informally justified simplification, performed in order to speed up execution of a model, preserves its original behaviour. A suitable small example seems the best way to examine this.

Consider a simple communications system, shown in Figure 6.10, where a terminal inputs data to create a *Stream* of frames and sends these to a *Transmitter*, where packets are built out of frames before being passed to an output stage which sends them down a channel.

Figure 6.10: Activity diagrams of network model

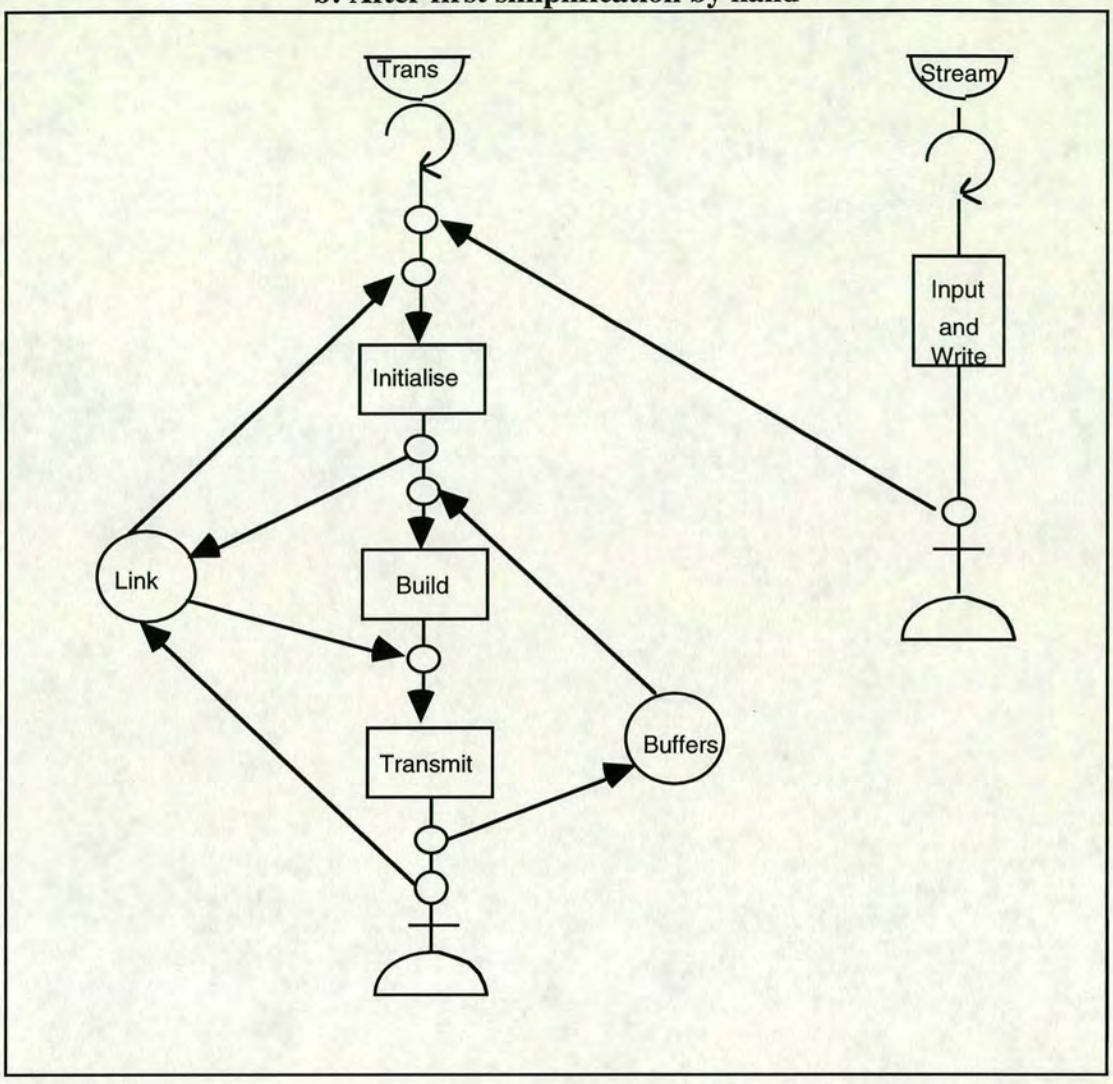
a: Before simplification by hand



This example contains two processes which might reasonably be modelled separately. For the purposes of simplification they will be considered one at a time. Since the only communication between them is through a schedule call from the *Stream* process to the *Trans* process this is reasonable.

The first simplification replaces the internal logic of the *Stream process* with a simple hold. This is justified intuitively by noting that it was originally made up of two sequential holds, with one bracketed by an acquire/release of a resource. As long as this resource is not already in use this should have no effect.

b: After first simplification by hand

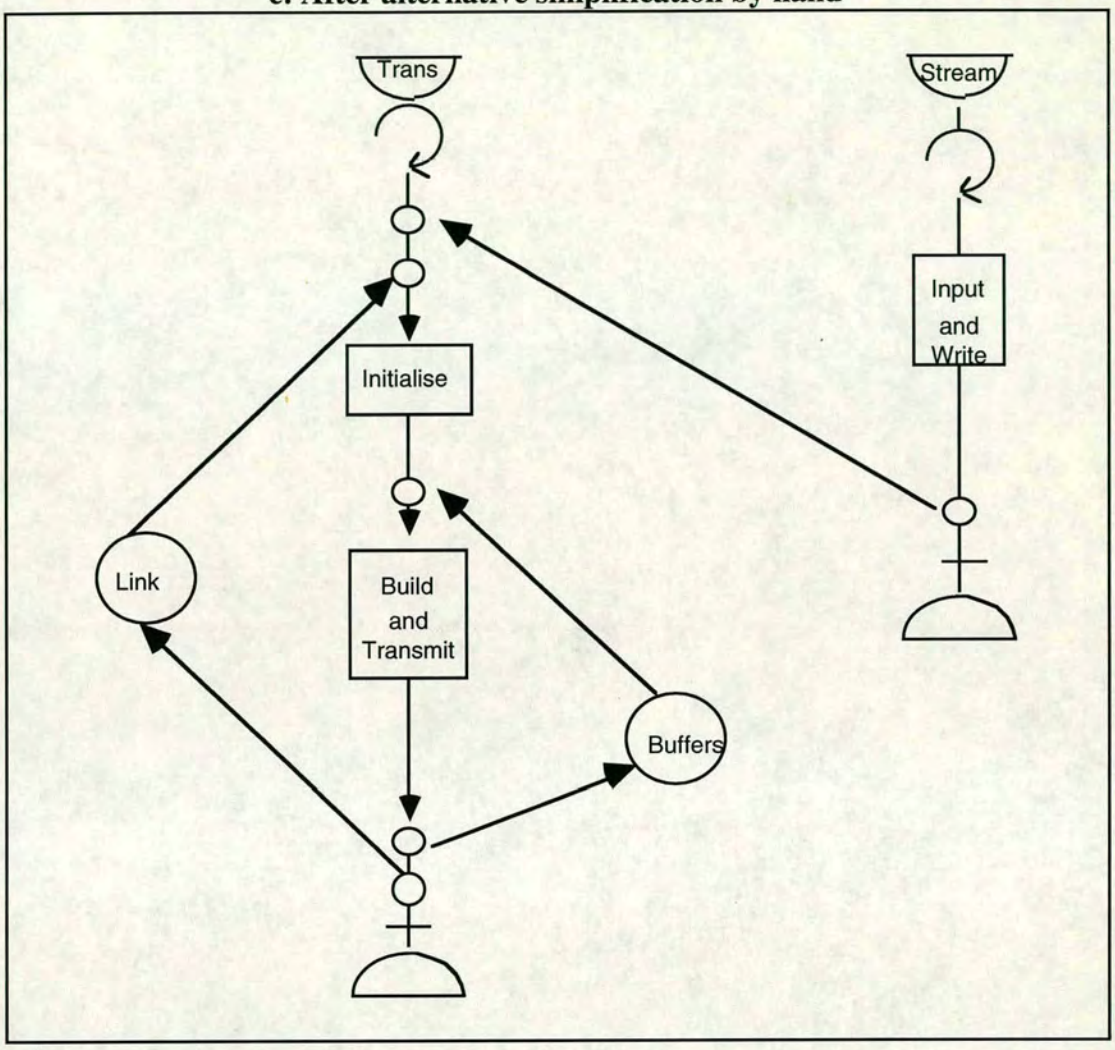


The second simplification applies the same sort of thinking to the *Stream process*. Since this releases the *Link* resource only to re-acquire it immediately after the *Build* phase of its operation, it might be safe to assume that it merely keeps the *Link* throughout.

To decide which if any of these simplifications is valid, from a behavioural point of view, corresponding CCS models were constructed and tested with the Concurrency Workbench. In writing these models the question of which resources were to be considered local and which global had to be answered and the decision was taken that the *Memory* resource would be considered as local to the Stream process, while the *Link* and *Buffers* resources would be global. This is reflected in the fact that

$memAcq_4$ and $memRel_4$ are restricted when the agent *Input* is defined, to represent a single *Stream* process and its *Memory*, in Figure 6.11a, while $linkAcq_1$, $linkRel_1$,

c: After alternative simplification by hand



$buffAcq_2$ and $buffRel_2$ are only restricted when two *Trans* processes are combined in $Model_b$.

Figure 6.11: CCS versions of models in figure 6.10

a: Full model

$Stream$	$\stackrel{\text{def}}{=}$	$(T_{Input})\ \overline{memAcq_4}\ (T_{Write})\ \overline{memRel_4}\ .\ \overline{tSched}\ .Stream$
Mem_4	$\stackrel{\text{def}}{=}$	$memAcq_4.Mem_0$
Mem_0	$\stackrel{\text{def}}{=}$	$memRel_4.Mem_4$
$Input$	$\stackrel{\text{def}}{=}$	$(Stream \mid Mem_4)\backslash\{memAcq_4, memRel_4\}$
$Model_a$	$\stackrel{\text{def}}{=}$	$(Input \mid Input)$
$Trans$	$\stackrel{\text{def}}{=}$	$tSched.\ \overline{linkAcq_1}\ .Starter$
$Starter$	$\stackrel{\text{def}}{=}$	$(T_{Init})\ \overline{buffAcq_2}\ .\ \overline{linkRel_1}\ .Builder$
$Builder$	$\stackrel{\text{def}}{=}$	$(T_{Build})\ \overline{linkAcq_1}\ .Transmitter$
$Transmitter$	$\stackrel{\text{def}}{=}$	$(T_{Transmit})\ \overline{buffRel_2}\ .\ \overline{linkRel_1}\ .Trans$
$Link_1$	$\stackrel{\text{def}}{=}$	$linkAcq_1.Link_0$
$Link_0$	$\stackrel{\text{def}}{=}$	$linkRel_1.Link_1$
$Buffs_2$	$\stackrel{\text{def}}{=}$	$buffAcq_2.Buffs_0$
$Buffs_0$	$\stackrel{\text{def}}{=}$	$buffRel_2.Buffs_2$
$Model_b$	$\stackrel{\text{def}}{=}$	$(Trans \mid Trans \mid Link_1 \mid Buffs_2)\backslash\{linkAcq_1, linkRel_1, buffAcq_2, buffRel_2\}$
$Model$	$\stackrel{\text{def}}{=}$	$(Model_a \mid Model_b)\backslash\{tSched\}$

b: First simplification by hand

$Stream$	$\stackrel{\text{def}}{=}$	$(T_{Input}+T_{Write})\ \overline{tSched}\ .Stream$
$Input$	$\stackrel{\text{def}}{=}$	$Stream$

The effect of this simplification is to leave the externally observable behaviour of the overall model unchanged. Since the *Memory* resource was totally private and only used sequentially, it could never lead to alternatives within the *Stream* agent. Since it cannot engage in external actions, it can be safely removed.

c: Alternative simplification by hand

$$
\begin{array}{lll}
Trans & \stackrel{def}{=} & tSched.\ \overline{linkAcq_1}\ .Starter \\
Starter & \stackrel{def}{=} & (T_{Init})\ \overline{buffAcq_2}\ .Builder \\
Builder & \stackrel{def}{=} & (T_{Build}+T_{Transmit})Transmitter \\
Transmitter & \stackrel{def}{=} & \overline{buffRel_2}\ .\ \overline{linkRel_1}\ .Trans
\end{array}
$$

The second modification is less successful. Since the *Trans* processes compete for the *Link* and *Buffs* resources, the removal of the releasing of the *Link* means that there is one less point where the other *Trans* process could acquire it. Where it does acquire it the contention leads to potential deadlock, which is lost in the simplified version. This is clearly a dangerous simplification. Testing with the Concurrency Workbench, as shown in Appendix C, makes this quite apparent.

6.3 Phenomena which cause problems

As well as wanting to obtain the simplest model with the desired behaviour, it is often important to know whether a model avoids certain problems. If not, knowing before executing the model may help in two ways. Firstly, the behaviour may be a correct representation of the behaviour of the system being modelled. In this case either it will help in setting up appropriate experiments using this model or, in cases where the simulation is being conducted to establish behavioural properties, will save costly simulation, which might not have revealed the problem anyway. Secondly, the behaviour of the model may not match that of the system and the pre-analysis then indicates a need to re-code the model to produce the correct behaviour. In both cases, a lot of unnecessary time can be saved and potentially misleading results avoided.

6.3.1 Simultaneous events

In Chapter 2 Schruben and Yücesan's rules for analysis of the structure of simulation nets were described. Schruben offers in his Rule 3 a way of identifying *possibly simultaneous events*. Such situations are at the heart of a number of problems with execution of interleaving actions in discrete event simulations. When two events occur together, the simulation must decide to let one proceed first, even though no

simulation time elapses, i.e. though the simulation clock does not advance. This is typically resolved by *branching probabilities*, by *establishing priorities* or by treating the situation as a *race condition*. Probabilities allow a choice of which action is allowed to be made according to some random drawing and an associated probability function, preventing any others. Priorities may be decided by the programmer or pre-defined, i.e. in Petri net simulators it is normal to allow timeless (Instantaneous) transitions to fire first. Race conditions allow the activity which would finish first to proceed and kill any others starting at the same time. Where the time to complete an activity is defined stochastically, this is effectively a probabilistic choice based on the relative rates of completion of the activities. It is easy enough to handle probabilistic choices in a simulation, once they are identified. The same is true of priorities. Most problems arise from events which are expected to be genuinely concurrent.

Genuinely simultaneous events

To see this problem in one manifestation, consider modelling the CSMA/CD level of an Ethernet. Simplifying this to one of its aspects, each station on an Ethernet is allowed to try to transmit as long as the net is free. The stations have the capability of sensing when this changes, by detecting the presence or absence of the carrier signal (*carrier sense medium access* or CSMA). If one station starts to transmit, others which subsequently wish to do so are forced to wait until the current transmission is complete. Thus there may be several stations blocked at the end of a transmission. Once the Ether is free they will all sense this and try to transmit, effectively, simultaneously. This results in their packets *colliding*, which they are also able to detect (*collision detect* or CD). When a collision occurs, all transmitting stations back off by an individually determined random interval, to minimise the chances of a further collision. Collisions can also occur where one station begins transmitting and is followed by a second before the first carrier has reached it. This cause of collision is not considered in what follows, for simplicity.

A system with this form of backoff seems simple enough to model, given the *Res*, *CondQ* and *WaitUntil* constructs in the process interaction paradigm. In Figure 4.9 an activity diagram for a suggested model was given. (In fact that also included a test for a maximum number of re-tries after collision before abandoning the packet, which is also not considered here for simplicity.) Unfortunately, the accurate representation of such a protocol is not as straightforward as it seems. Some

alternative models, all of which are based on genuine attempts by modellers, for this situation are considered below in an attempt to isolate the potential causes of confusion. Then, using the CCS equivalents of these models, the contribution that could be made by behavioural analysis prior to simulation is assessed.

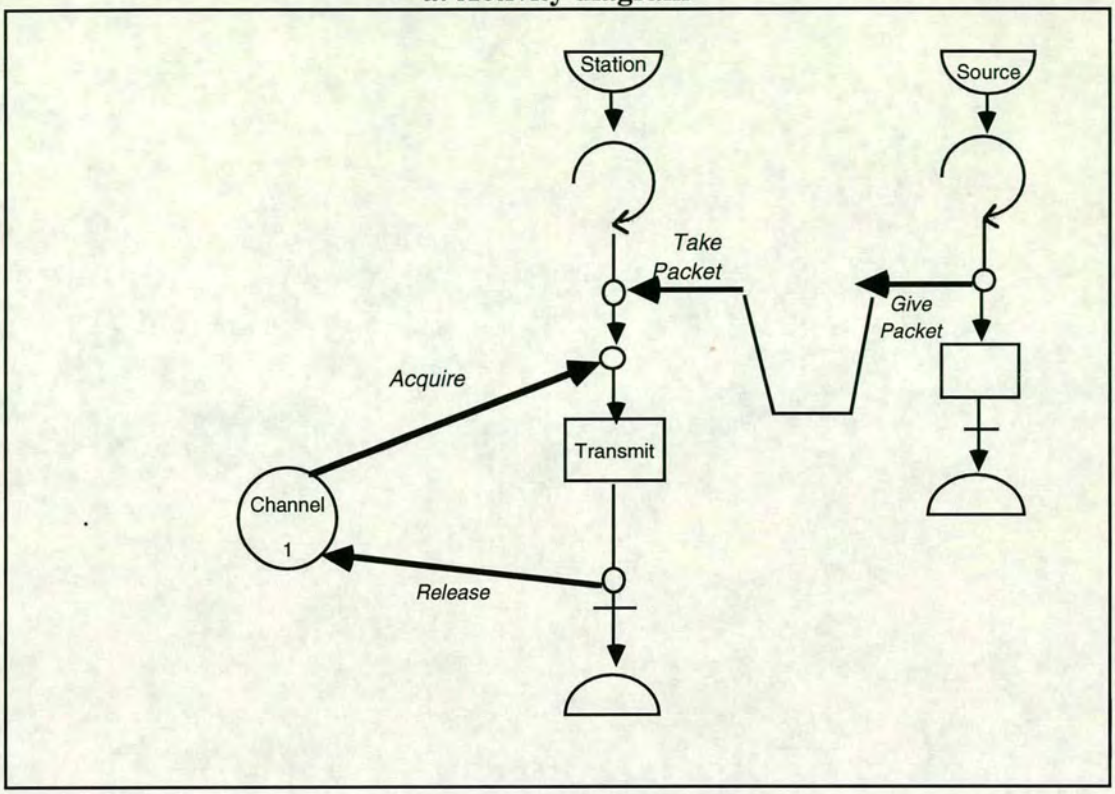
A naïve model using just Res

Many modellers have fallen into the obvious trap of treating Ethernet as a simple resource contention problem. Thus they model the channel as a *Res* with amount 1 and have stations competing to acquire it, as shown in the activity diagram in Figure 6.12. This model has the advantage of simplicity, but inevitably causes a problem, since the first station in the queue for the Res will always get to transmit, without the others getting to try.[1]

[1] The DEMOS versions and corresponding traces of these models are found in Appendix B.

Figure 6.12: Activity diagram and CCS of naïve Ethernet as Res model

a: Activity diagram



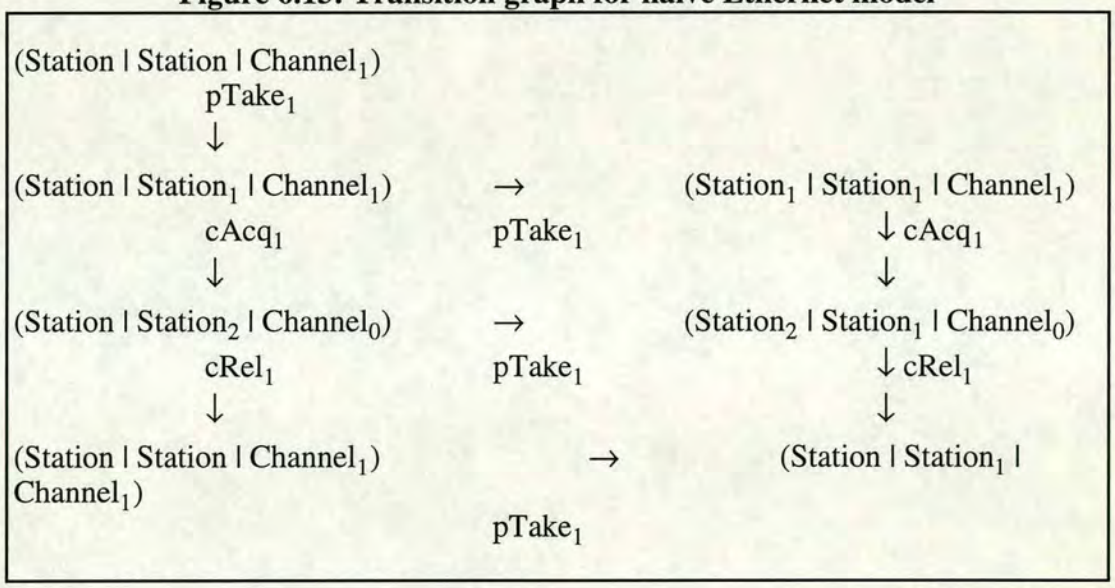
b: the CCS model

Station	$\stackrel{\text{def}}{=}$	$\overline{inQTake_1}\,.\,\overline{eAcq_1}\,.Sending$
Sending	$\stackrel{\text{def}}{=}$	$(T_{Transmit})\,\overline{eRel_1}\,.Station$
Source	$\stackrel{\text{def}}{=}$	$(T_{Arrival})\,\overline{inQGive_1}\,.Source$
InQ_0	$\stackrel{\text{def}}{=}$	$inQGive_1.InQ_1$
InQ_n	$\stackrel{\text{def}}{=}$	$inQGive_1.InQ_{n+1} + inQTake_1.InQ_{n-1}$ $0<n<$Maxint
InQ_{Maxint}	$\stackrel{\text{def}}{=}$	$inQTake_1.InQ_{Maxint-1}$
Transmitter	$\stackrel{\text{def}}{=}$	$(Station \mid Source \mid InQ_0)\backslash\{inQTake_1, inQGive_1\}$
$Ether_1$	$\stackrel{\text{def}}{=}$	$eAcq_1.Ether_0$
$Ether_0$	$\stackrel{\text{def}}{=}$	$eRel_1.Ether_1$
Model	$\stackrel{\text{def}}{=}$	$\left(Ether_1 \mid \prod_{i=1}^{NStations} Transmitter\right)\backslash\{eAcq_1, eRel_1\}$

Running the DEMOS model, the trace reveals that one of the stations continues with its transmission and the others are blocked, rather than them all backing off. This is as one might expect from an analysis of the DEMOS *Res* mechanisms. It is also important to notice that this is not dependent on the queueing version in unmodified DEMOS, but still applies to *modified* DEMOS.

Simplifying the model to its key elements, by focusing on the Station and Channel processes and assuming that there is always a packet to transmit, and assuming two Stations, a transition graph can be derived which helps to show what is restricting the behaviour of the system in undesirable ways. Transmission is only possible when a Station has reached $Station_2$. This is never the case for both Stations at the same time.

Figure 6.13: Transition graph for naïve Ethernet model



Introducing a *CondQ* to model concurrent behaviour

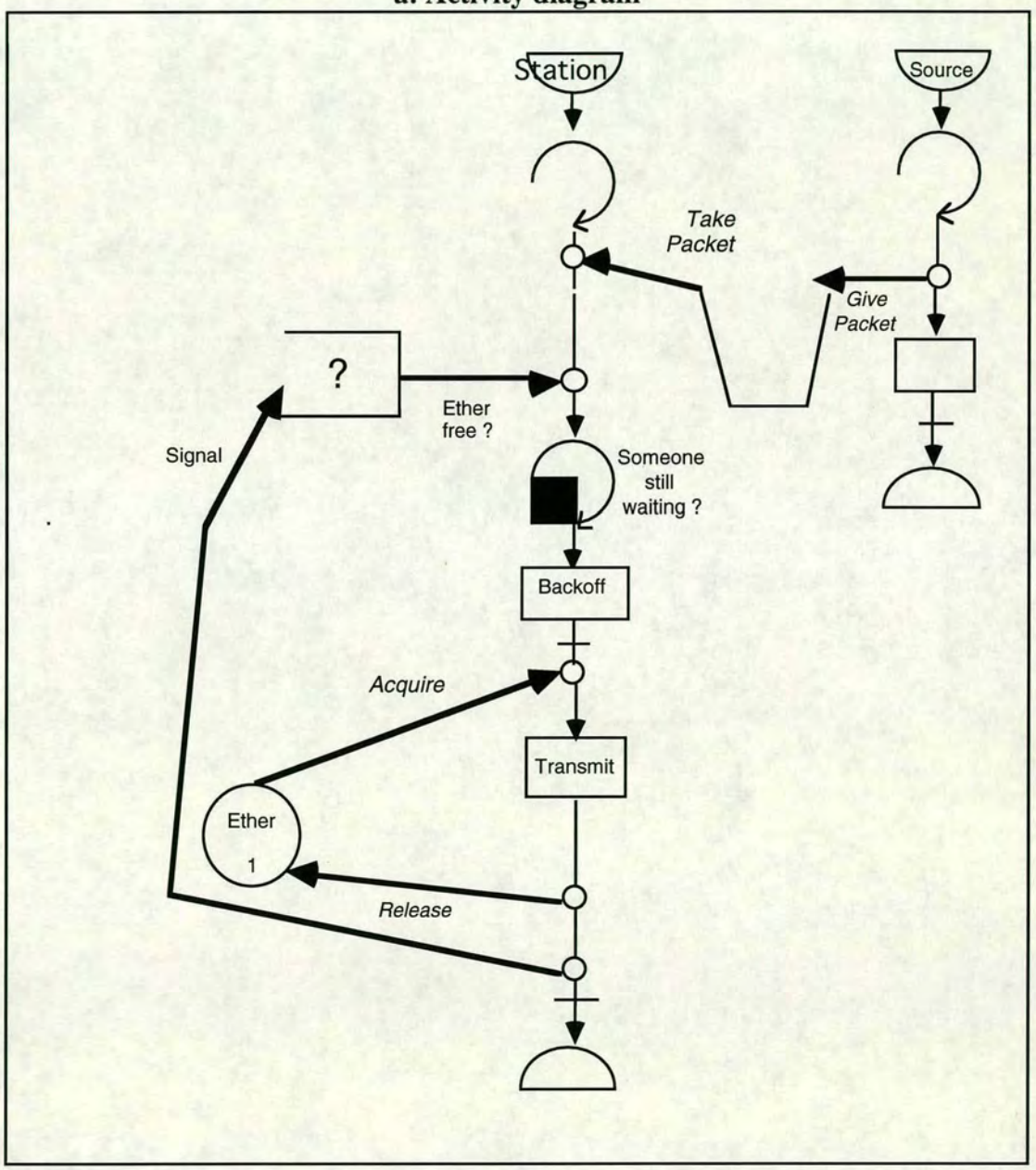
Instead of simply using a resource, a CondQ could be used, as shown in Figure 6.13, to hold Station processes until the Ether becomes free[1]. Once a transmitting Station has finished, it is responsible for signalling that the Ether is free. Then, before acquiring the Ether Res, each Station in turn can check if the length of the queue for the Ether is greater than one. If so, a collision has occurred and the Station should back off. Unfortunately this still depends on a queue (this time a CondQ), each of

[1] Only those parts of the CCS models which are new or have changed are shown in Figures 6.15 and 6.16.

whose members is scheduled in turn, thereby removing itself from that queue, before testing for collision. The CondQ's length decreases each time a Station leaves it, until the last Station finds the number remaining has reached zero and proceeds to transmit. Collision, shown by a backoff, now happens for all but one of the Stations, which is still not quite what is wanted. This has a similar restriction, shown by its transition graph, to the model using just the resource.

Figure 6.14: CondQ used to model Ethernet

a: Activity diagram



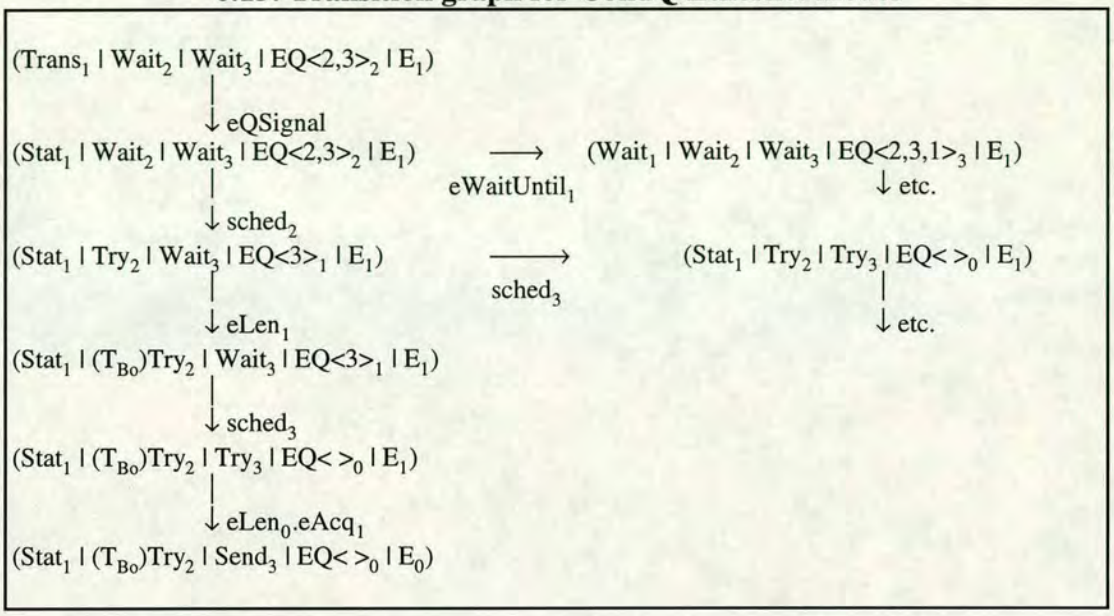
b: The CCS

$Station_{id}$	$\stackrel{def}{=}$	$\overline{inQTake_1}\,.\,\overline{eWaitUntil_{id}}\,.Waiting_{id}$
$Waiting_{id}$	$\stackrel{def}{=}$	$sched_{id}.Trying_{id}$
$Trying_{id}$	$\stackrel{def}{=}$	$\sum_{i=1}^{NStations}(eLen_i(T_{Backoff})Trying_{id}) + (eLen_0.\,\overline{eAcq_1}\,.Sending_{id})$
$Sending_{id}$	$\stackrel{def}{=}$	$(T_{Transmit})\,\overline{eRel_1}\,.Done_{id}$
$Done_{id}$	$\stackrel{def}{=}$	$\overline{eQSignal}\,.Station_{id}$
$Source$	$\stackrel{def}{=}$	$(T_{Arrival})\,\overline{inQGive_1}\,.Source$
InQ_0	$\stackrel{def}{=}$	$inQGive_1.InQ_1$
InQ_n $0<n<Maxint$	$\stackrel{def}{=}$	$inQGive_1.InQ_{n+1} + inQTake_1.InQ_{n-1}$
InQ_{Maxint}	$\stackrel{def}{=}$	$inQTake_1.InQ_{Maxint-1}$
$Transmitter_{id}$	$\stackrel{def}{=}$	$(Station_{id} \mid Source \mid InQ_0)\backslash\{inQTake_1, inQGive_1\}$
$Ether_1$	$\stackrel{def}{=}$	$eAcq_1.Ether_0$
$Ether_0$	$\stackrel{def}{=}$	$eRel_1.Ether_1$
$EtherQ\langle L\rangle_{len}$	$\stackrel{def}{=}$	$eWaitUntil_n.EtherQ\langle L,n\rangle_{len+1} + eQSignal.Signal\langle L\rangle_{len}$
$Signal\langle\,\rangle_{len}$	$\stackrel{def}{=}$	$EtherQ\langle\,\rangle_{len}$
$Signal\langle h,L\rangle_{len}$	$\stackrel{def}{=}$	$\overline{sched_h}\,.Signal\langle L\rangle_{len-1} + \overline{eLen_{len}}\,.Signal\langle h,L\rangle_{len}$
$Model$	$\stackrel{def}{=}$	$\left(Ether_1 \mid EtherQ\langle\,\rangle_0 \mid \prod_{i=1}^{NStats} Station_i\right)\backslash L(Model)$

The transition graph in Figure 6.15 shows the relevant part of this model's behaviour for a three *Station* model. It starts from the state where one *Station* has just finished transmission and released the *Ether* resource, while the other two are waiting for this to happen. The first to go backs off and the second proceeds. There are two side branches, marked *etc.*, which are ignored. The first is the case where the *Station* which has just finished tries immediately to transmit another packet and enters the CondQ, taking its length to three. It is easy to show that the same possibilities result, except that now the original two waiting *Stations* back off, while the one that has just finished sends again. The other branch shows both the waiting *Stations* being

scheduled before either tests the CondQ's length. This is not the behaviour of the DEMOS model, since the effect of one process scheduling another is to place it in the event list behind the current one, not to preempt the current one. It was not felt worth the extra complexity of preventing this in the current model.

6.15: Transition graph for CondQ Ethernet model



Schruben's rules and this model

If this mechanism is modelled using Schruben's simulation nets, described in Chapter 2, the same problem re-asserts itself. Schruben's Rule 3 is: *Event scheduling priorities are required when the intersection of the state variable sets of two vertices is non-empty*. This fails to distinguish the possibility of multiple competing instances of the same event, which is allowed in the extended Simulation Graph formalism through parameterised edges carrying process identifiers, from simple contention. If the graphs are unrolled, so that each process instance is separately represented, Rule 3 identifies that there is a possible problem, but does not identify what it is. Schruben's rules are really concerned with tie-breaking rather than concurrent events.

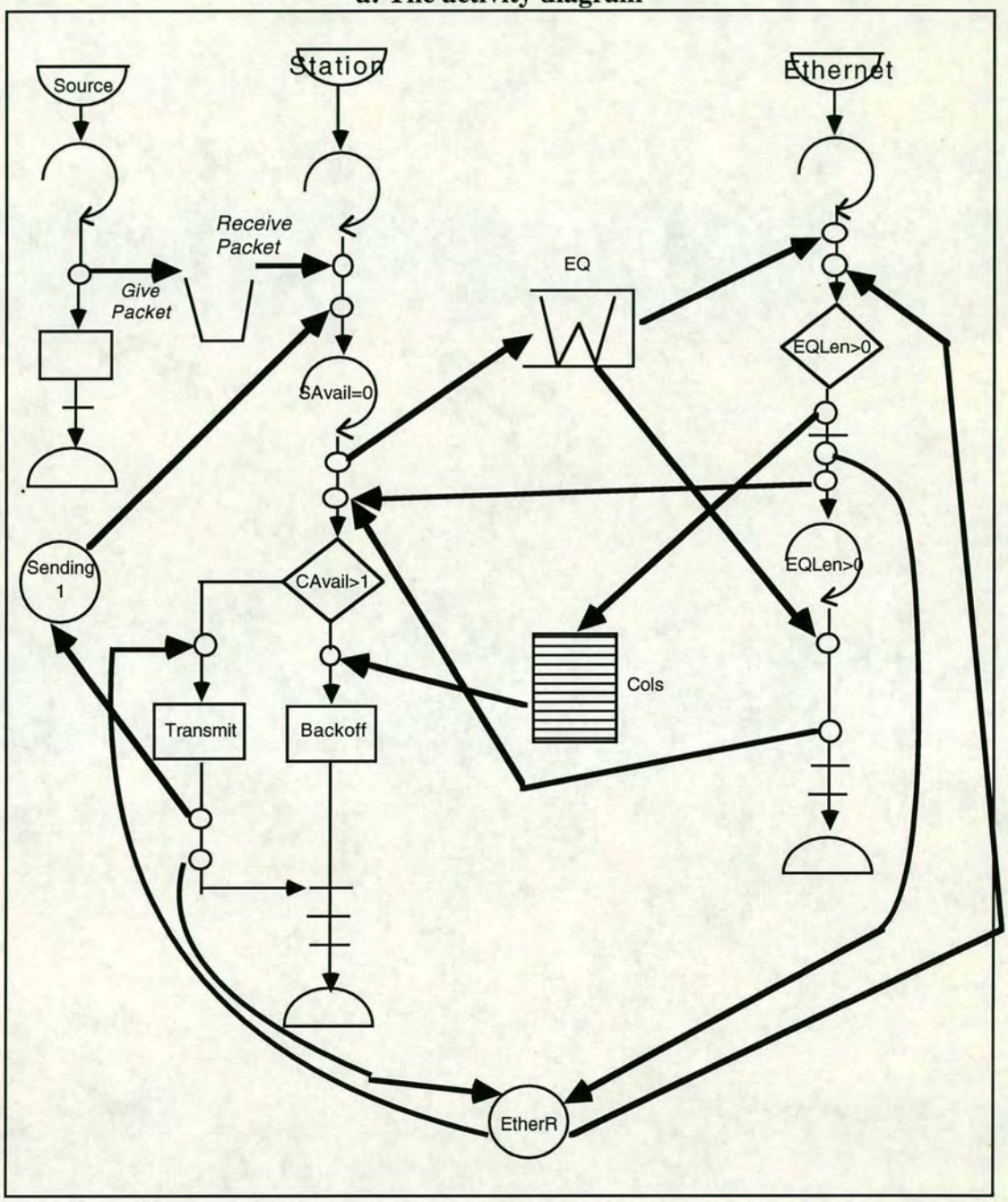
A "correct" model of CSMA/CD behaviour

To allow true concurrency to be represented in this case, the model needs to be reformulated so that the Ethernet is an active process, which can co-opt the waiting stations and set them to backoff if more than one wishes to proceed. This involves a flag within the *Station* process, which records when a transmission has been successful, and a means of indicating to a re-scheduled *Station* whether a collision

has occurred. In the model in Figure 6.16 a Res and a Store are used for this, but these are strictly used as a Boolean and an integer, respectively.

Figure 6.16: A correctly behaving Ethernet model

a: The activity diagram



b: The CCS model

$$Station_{id} \stackrel{def}{=} \overline{iQTake_1}\,.\,\overline{sAcq_1}\,.Trying_{id}$$

$$Trying_{id} \stackrel{def}{=} sAvail_0.\,\overline{eQWait_{id}}\,.Waiting_{id} + sAvail_1.Station_{id}$$

$$Waiting_{id} \stackrel{def}{=} eSched_{id}.\left(cAvail_0.\,\overline{etAcq_1}\;(T_{Transmit})Done_{id} + \sum_{i=1}^{NStats} cAvail_i.BackOff_{id}\right)$$

$$Done_{id} \stackrel{def}{=} \overline{sRel_1}\,.\,\overline{etRel_1}\;Trying_{id}$$

$$BackOff_{id} \stackrel{def}{=} \overline{cRem_1}\;(T_{Backoff})Trying_{id}$$

$$Ethernet \stackrel{def}{=} eQCoopt_{id}.\,\overline{etAcq_1}\,.Used_{id}$$

$$Used_{id} \stackrel{def}{=} eQLen_0.Next + \sum_{i=1}^{NStats} eQLen_i.\,\overline{cAdd_{i+1}}\,).Next_{id}$$

$$Next_{id} \stackrel{def}{=} \overline{etRel_1}\,.\,\overline{eSched_{id}}\,.ReSched$$

$$ReSched \stackrel{def}{=} eQLen_0.Ethernet + \sum_{i=1}^{NStats} eQLen_i.eQCoopt_{id}.\,\overline{eSched_{id}}\,.ReSched$$

$$Sending_1 \stackrel{def}{=} sAcq_1.Sending_0 + \overline{sAvail_1}\,.Sending_1$$

$$Sending_0 \stackrel{def}{=} sRel_1.Sending_1 + \overline{sAvail_0}\,.Sending_0$$

$$Cols_0 \stackrel{def}{=} \sum_{i=1}^{NStats} cAdd_i.Cols_i + \overline{cAvail_0}\,.Cols_0$$

$$Cols_n \stackrel{def}{=} \sum_{i=1}^{NStats-n} cAdd_i.Cols_{i+n} + cRem_1.Cols_{n-1} + \overline{cAvail_n}\,.Cols_n \qquad NStats \geq n > 0$$

$$Cols_{NStats} \stackrel{def}{=} cRem_1.Cols_{NStats-1} + \overline{cAvail_{NStats}}\,.Cols_{NStats}$$

$$EQ<>_0 \stackrel{def}{=} eQWait_n.EQ\langle n\rangle_1 + \overline{eQLen_0}\,.EQ<>_0$$

$$EQ\langle n,L\rangle_i \stackrel{def}{=} eQWait_m.EQ\langle n,L,m\rangle_{i+1} + \overline{eQCoopt_n}\,.EQ\langle L\rangle_{i-1} + \overline{eQLen_i}\,.EQ\langle n,L\rangle_i$$

L is any list of unique integers in 0..NStats, *n* not in *L*, *k* not in *L*, NStats≥*n* >0, NStats≥*k* >0

$$IQ_0 \stackrel{def}{=} iQGive_1.InQ_1$$

$$IQ_n \stackrel{def}{=} iQGive_1.InQ_{n+1} + iQTake_1.InQ_{n-1} \qquad 0 < n \leq Maxint$$

$$IQ_{Maxint} \stackrel{def}{=} iQTake_1.InQ_{Maxint-1}$$

$$Sender \stackrel{def}{=} (T_{Arr})\,\overline{iQGive_1}\,.Sender$$

$$EtherR_1 \stackrel{def}{=} etAcq_1.etRel_1.EtherR_1$$

$$Transmitter_{id} \stackrel{def}{=} (Station_{id} \mid Sender \mid IQ_0 \mid Sending_1) \setminus \{sAcq_1, sRel_1, iQGive_1, iQTake_1, sAvail_1, sAvail_0\}$$

$$Model \stackrel{def}{=} \left(Ethernet \mid \prod_{i=1}^{NStats} Transmitter_i \mid EtherR_1 \mid Cols_0 \mid EQ{<>}_0 \right) \setminus L(Model)$$

This model is much more complicated than its unsuccessful predecessors. Although it still reduces to a fairly compact DEMOS model, as shown in Appendix B, its logic requires some careful analysis. From this it is possible to find some general characteristics which are necessary for a process based discrete event model to show genuine concurrency. Unfortunately the current version of the Concurrency Workbench could not analyse the complete model, but it did provide useful feedback in the form of the reachable states of the components and in simulating the outcomes for the model in the relevant regions of its transition graph.

As with the simpler models, the Source is irrelevant to the behaviour in which we are interested, simply imposing an occasional delay at the start of a *Station* process. Internally the *Station* uses the *Sending* resource to keep track of whether it has just transmitted or not, i.e. as a Boolean flag. This sets the stopping condition of the inner loop in the *Station*. As long as it is trying to transmit, the *Station* first waits for the *Ethernet* process to set up the correct conditions and then follows either a transmit branch or a backoff branch, depending on the state of the *Cols* store. The *EtherR* resource controls whether a *Station* or the *Ethernet* proceeds, i.e. prevents the *Ethernet* scheduling further *Stations* while one is transmitting. It has no effect if the *Stations* have to back off.

The *Ethernet* is only active when there are *Stations* in the *EQ*, i.e. waiting to transmit, and when the *EtherR* resource is free. This is essentially the carrier sense aspect of CSMA/CD. Once it becomes active, the *Ethernet* enacts the collision detect aspect of the protocol, checking how many *Stations* are currently waiting in the *EQ*, setting a flag for each of them accordingly, by adding that number to *Cols*, and scheduling all of them. Once it has done this the *Ethernet* gives away control by releasing the *EtherR* resource and waiting in the *EtherQ* for new *Stations* to arrive for transmission and then for the *EtherR* resource to be free.

This intricate mechanism seems at first sight too specific to the CSMA/CD protocol to be capable of generalisation. On closer examination, however, it reveals some essential requirements for a general mechanism, which are:

> The model must be able to reach a state where more than one process could perform the same next action.
>
> The case where only one such process currently exists must be differentiated from the case where many are ready, by a test which gives the same, correct answer to every ready process, in a manner which will not be altered when others begin to act.
>
> Each process must now act according to the test result, allowing the others to do the same for as long as their independent, concurrent activities last.

While there may be many special cases where this could be done differently, the central scheduling process shown here is a general solution. The Wait Queue (*EQ*) performs the task of blocking potential actors until the scheduler is free to proceed. The Res (*EtherR*) blocks and releases the scheduling agent according to the conditions for the action of interest being allowed. The Store (*Cols*) provides a flag for each agent awoken by the scheduler as well as a flag to differentiate the single agent case. Indeed it is tempting to add such a primitive to *modified* DEMOS. It is important to note that no new arrivals can be allowed in the Wait Queue once the scheduler has gained control and made the test of the concurrency level. This is easily enforced, since the scheduler does not cause time to advance or in any other way yield control until it returns to the co-opting side of the Wait Queue itself.

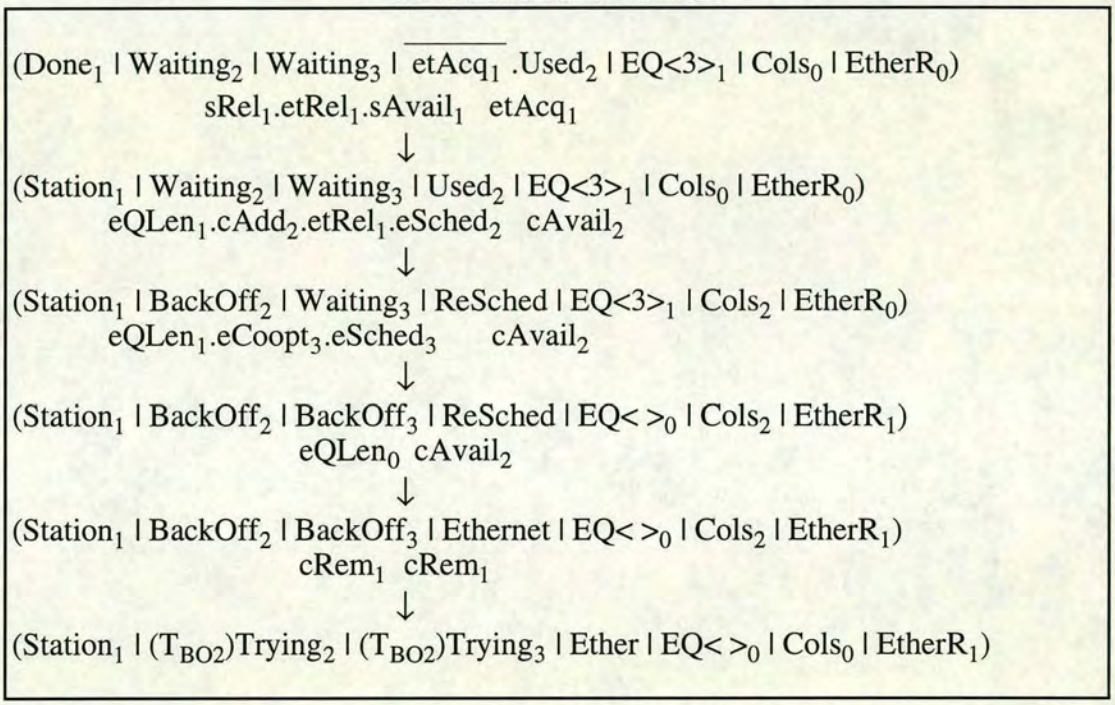
But this depends on general reasoning about the model. CCS has provided no direct answers so far, except to show why the simple resource model and the CondQ model failed. However, those cases showed an important test that can be applied to determine whether a model can mimic concurrency. If there are no states reachable in the model where all those agents which should be able to perform an action simultaneously have that action as their next one, the model is not adequate. Figure 6.17 shows parts of the transition diagram for the successful model, where all (both in this case) agents can proceed to backoff (6.17a), but transmission takes place when only one is initially waiting (6.17b). A further possibility is that at the start of 6.17a the third *Station* agent reaches the *Waiting* state before the *Ethernet* agent

checks the *EQ*. This would happen where the third *Station* had a packet waiting before it ended its previous transmission (6.17c).

All of the checking of the logic of this model was greatly assisted by the use of CCS. Simply producing a consistent model clarified many problems. Getting the model accepted by the Concurrency Workbench provided further checking of the model's consistency. Once entered, the use of the States command, to find the total state set, and the sim command, to follow the paths shown in Figure 6.17, allowed detailed debugging.

Figure 6.17: Transition diagram for modelling of true concurrency

a: Collision and backoff



b: Successful transmission

$$(Done_1 \mid Station_2 \mid Waiting_3 \mid \overline{etAcq_1}.Used_3 \mid EQ<>_0 \mid Cols_0 \mid EtherR_0)$$
$$sRel_1.etRel_1.sAvail_1 \quad etAcq_1$$
$$\downarrow$$
$$(Station_1 \mid Station_2 \mid Waiting_3 \mid Used_3 \mid EQ<>_0 \mid Cols_0 \mid EtherR_0)$$
$$eQLen_0.etRel_1.eSched_3 \quad cAvail_0.etAcq_1$$
$$\downarrow$$
$$(Station_1 \mid Station_2 \mid (T_{Trans3})Done_3 \mid ReSched \mid EQ<>_0 \mid Cols_0 \mid EtherR_0)$$
$$eQLen_0$$
$$\downarrow$$
$$(Station_1 \mid BackOff_2 \mid (T_{Trans3})Done_3 \mid Ethernet \mid EQ<>_0 \mid Cols_0 \mid EtherR_0)$$

c: Immediate re-transmission, collision and backoff

$$(Done_1 \mid Waiting_2 \mid Waiting_3 \mid \overline{etAcq_1}.Used_2 \mid EQ<3>_1 \mid Cols_0 \mid EtherR_0)$$
$$sRel_1.etRel_1.sAvail_1.iQTake_1.sAcq_1 \quad etAcq_1$$
$$\downarrow$$
$$(Trying_1 \mid Waiting_2 \mid Waiting_3 \mid Used_2 \mid EQ<3>_1 \mid Cols_0 \mid EtherR_0)$$
$$sAvail_0.eQWait_3$$
$$\downarrow$$
$$(Waiting_1 \mid Waiting_2 \mid Waiting_3 \mid Used_2 \mid EQ<3,1>_2 \mid Cols_0 \mid EtherR_0)$$
$$eQLen_2.cAdd_3.etRel_1.eSched_2 \quad cAvail_3$$
$$\downarrow$$
$$(Waiting_1 \mid BackOff_2 \mid Waiting_3 \mid ReSched \mid EQ<3,1>_2 \mid Cols_2 \mid EtherR_0)$$
$$eQLen_2.eCoopt_3.eSched_3 \quad cAvail_3$$
$$\downarrow$$
$$(Waiting_1 \mid BackOff_2 \mid BackOff_3 \mid ReSched \mid EQ<1>_1 \mid Cols_2 \mid EtherR_1)$$
$$eQLen_1.eCoopt_1.eSched_1 \quad cAvail_3$$
$$\downarrow$$
$$(BackOff_1 \mid BackOff_2 \mid BackOff_3 \mid ReSched \mid EQ<>_0 \mid Cols_2 \mid EtherR_1)$$
$$eQLen_0 \quad cAvail_3$$
$$\downarrow$$
$$(Station_1 \mid BackOff_2 \mid BackOff_3 \mid Ethernet \mid EQ<>_0 \mid Cols_2 \mid EtherR_1)$$
$$cRem_1 \quad cRem_1 \quad cRem_1$$
$$\downarrow$$
$$((T_{BO1})Trying_1 \mid (T_{BO2})Trying_2 \mid (T_{BO2})Trying_3 \mid Ether \mid EQ<>_0 \mid Cols_0 \mid EtherR_1)$$

It had been anticipated that the modal μ-calculus could be employed to answer some of these questions, but in practice it seemed poorly adapted to making general queries such as:

if a state is reached where two or more agents are in the Wait Queue, is it possible for an agent to reach the broadcasting state before all of those in the queue have backed off.

This may be due to poor understanding of the capabilities of modal logics and remains an open question.

Races

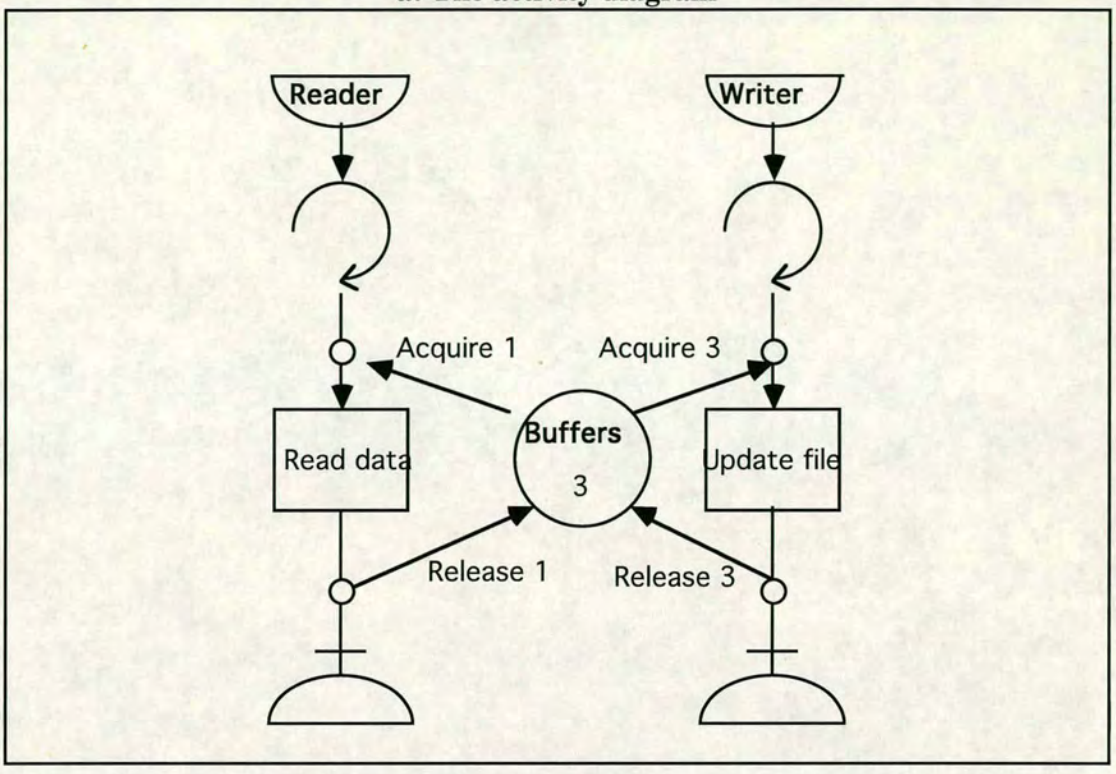
A race occurs where more than one activity can be under way at the same time, but where only the first to complete will actually be deemed to have succeeded. An example might be where several packets enter a packet switched network, but where only the first one to reach the destination node will be accepted, the others being lost. This is different from the idea of simultaneously acting events, since the time of completion determines which is deemed to have occurred. It seems unlikely that CCS or any discrete time variant will answer many meaningful questions, apart from cases where a deterministic delay is involved for all racing processes. In all cases, the same condition for a race being able to happen applies as for concurrent activities above. A test on the duration of each event could then determine which one proceeds and which ones die, but this requires an extension to the semantics of CCS. It would be more appropriate to consider PEPA or a similar stochastic process algebra for such cases.

6.3.2 Starvation

The reader/writer model of Chapter 4 shows an example of a resource used to enforce mutual exclusion. This can also be used to implement a semaphore. Under appropriate timings this model can produce starvation. Figure 6.18 shows the mapping into TCCS for that model.

Figure 6.18: Reader writer model as an example of potential starvation

a: The activity diagram



b: The CCS

Reader	$\stackrel{def}{=}$	$\delta.\ \overline{buffAcq_1}\ (T_{read})Thinker$
Thinker	$\stackrel{def}{=}$	$\overline{buffRel_1}\ (T_{think})Reader$
Writer	$\stackrel{def}{=}$	$\delta.\ \overline{buffAcq_3}\ (T_{update})Updater$
Updater	$\stackrel{def}{=}$	$\overline{buffRel_3}\ (T_{search})Writer$
$Buffs_3$	$\stackrel{def}{=}$	$\delta.buffAcq_1.Buffs_2 + \delta.buffAcq_3.Buffs_0$
$Buffs_2$	$\stackrel{def}{=}$	$\delta.buffAcq_1.Buffs_1 + \delta.buffRel_1.Buffs_3$
$Buffs_1$	$\stackrel{def}{=}$	$\delta.buffRel_1.Buffs_2$
$Buffs_0$	$\stackrel{def}{=}$	$\delta.buffRel_3.Buffs_3$
Model	$\stackrel{def}{=}$	$(Reader \mid Reader \mid Writer \mid Buffs_3)\backslash\{buffAcq_1, buffAcq_3, buffRel_1, buffRel_3\}$

Consider the *Reader* process. This is a simple cyclical process, defined in CCS by a right recursion. It requires only one buffer to proceed. The *Writer* process is structurally similar, but needs to acquire all the buffers before it can update them. This simple mutual exclusion example is interesting since it may induce starvation of the *Writer* by the *Reader* processes if the timings of the *Readers* are unfavourable. The resource is modelled as usual and is simplified as before. Finally the model is a parallel composition of all processes

Since there are only two *Reader* processes and only in them can a $buffAcq_1$ take place, and the only way to reach a $Buffs_0$ state is following a $buffAcq_3$, the only possible action of a $Buffs_0$ agent is a $buffRel_3$. Thus the graph of *Model* has two sub-graphs, which are only joined by the start state.

The problem of starvation may be summarised as the situation where, although it is theoretically possible to reach an agent (or sub-graph of the transition graph) within a model, under certain timing and priority or resource conditions, created when the other has proceeded, this cannot happen. Unlike the more general notion of *unfairness*, without timing information the best that can be said is that the *possibility* does or does not exist, i.e. that there is a choice from which two or more disjoint sub-agents start and at least one of them contains a cycle which can prevent return to the choice.

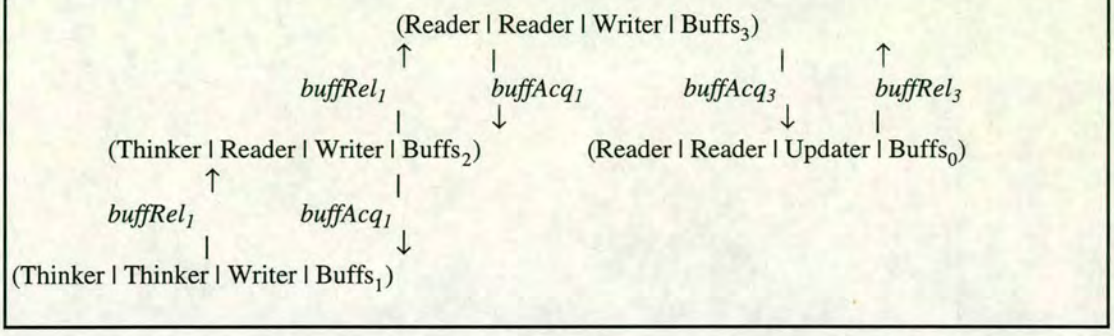
In the model above, this is clearly the start agent, *Model*. The two sub-agents *Reader* and *Writer* both cycle back to this choice, but *Reader* may remain within an internal cycle of activity. This is not strictly the same as livelock, since progress may be made by the overall system, even though part of it is starved. Working without timings the reachability graph of Figure 6.19 is produced.

It would be comparatively simple to phrase a question in the modal m-calculus of the form, "Is it possible for the model to reach a state (or perform an action) in the Writer cycle once it has reached (performed) one in the Reader inner cycle?" One such question is written in the Workbench syntax as:

```
bi X (Thinker| Thinker| Writer| Buffs1)\{buffAcq1,buffAcq3,buffRel1,buffRel3}
cp X min (X.<buffAcq3>T | <->X)
```

Thus, once the structure of the model is apparent, an answer can be expected. It is still perhaps reasonable to expect a modeller to be able to do this.

Figure 6.19: Reader/Writer reachability graph without timings

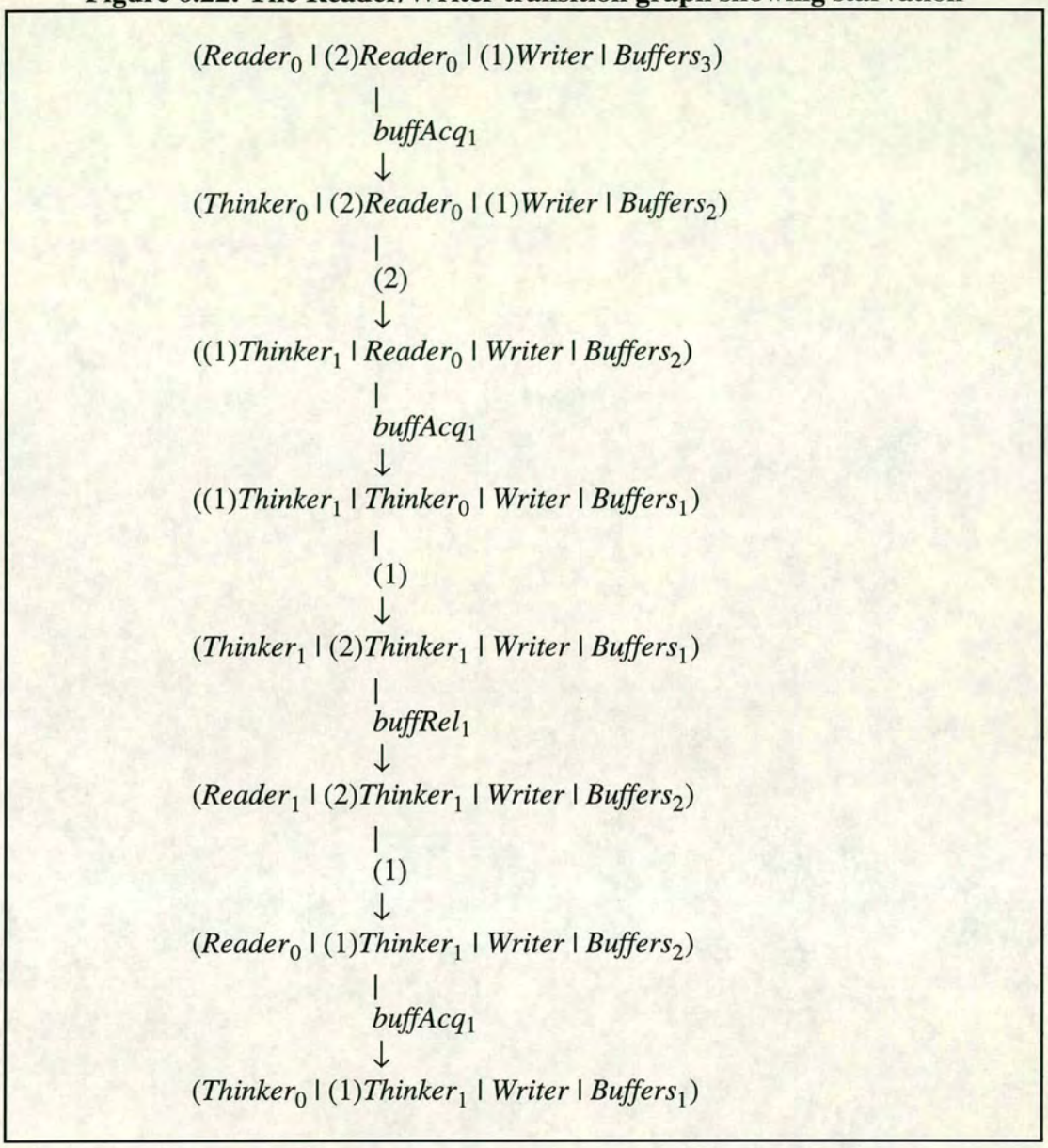


It is the secondary cycle between the two reader processes that prevents the writer from engaging in any activity. If timings are added which force the model into bad behaviour, the temporal version of CCS can be used to show this, as shown in Figure 6.21. The timings in the *Writer* agent are unimportant, as it will never be allowed to start as long as both *Reader*s do not release their buffers simultaneously. The *Reader* agent is extended into a series of sub-agents corresponding to time advancing. The overall model uses time prefixes to schedule the various *Reader*s and *Writer*s out of time with each other. The transition graph is now as shown in Figure 6.22.

Figure 6.21: Reader/Writer TCCS with timings forcing starvation

$Reader_0$	$\stackrel{def}{=}$	$\delta.\ \overline{buffAcq_1}\ .\ Thinker_0$
$Thinker_0$	$\stackrel{def}{=}$	$(3)Thinker_1$
$Thinker_1$	$\stackrel{def}{=}$	$\overline{buffRel_1}\ .\ Reader_1$
$Reader_1$	$\stackrel{def}{=}$	$(1)Reader_0$
$Model$	$\stackrel{def}{=}$	$(Reader_0 \mid (2)Reader_0 \mid (1)Writer \mid Buffers_3)\backslash \mathbb{L}(Model)$

Figure 6.22: The Reader/Writer transition graph showing starvation



The last state is identical, when re-ordered, to an earlier state and so the model will cycle indefinitely without *Writer* ever acting.

Expressing starvation

The property that starvation may be possible can be given in English as follows.

> Given a choice state, generated by applying the expansion theorem to the parallel composition of two agents, there is, from that state of the model, a path which may revisit that choice, but need not do so. If timing information or priorities are

added, it is possible to show cases where such a system will definitely behave badly.

6.3.4 Deadlock

The most widely known liveness property is probably deadlock. It is clearly capable of being represented in CCS, as noted in Chapter 2. Here the use of *modified* DEMOS and of CCS is shown to detect deadlock correctly in the harbour model.

Formalising the proof for the harbour model

To show whether deadlock is possible, irrespective of timings, and to explore why the Workbench gives different results to DEMOS, we initially use the simplified model used to show transition elimination above. This simplifies the analysis and is also important to an understanding of why DEMOS fails to behave in the way predicted. Together these changes give the model in Figure 6.22.

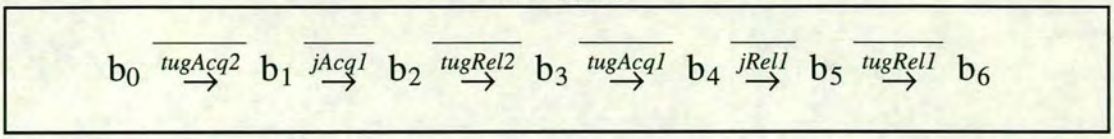
Figure 6.22: Harbour CCS model to show deadlock

$BOAT$	$\stackrel{def}{=} \overline{jAcq_1} \,.\, \overline{tugAcq_2} \,.\, \overline{tugRel_2} \,.\, \overline{tugAcq_1} \,.\, \overline{tugRel_1} \,.\, \overline{jRel_1}$
$TUGS_2$	$\stackrel{def}{=} (tugAcq_1.TUGS_1) + (tugAcq_2.TUGS_0)$
$TUGS_1$	$\stackrel{def}{=} (tugAcq_1.TUGS_0) + (tugRel_1.TUGS_2)$
$TUGS_0$	$\stackrel{def}{=} (tugRel_1.TUGS_1) + (tugRel_2.TUGS_2)$
$JETTIES_2$	$\stackrel{def}{=} (jAcq_1.JETTIES_1)$
$JETTIES_1$	$\stackrel{def}{=} (jAcq_1.JETTIES_0) + (jRel_1.JETTIES_2)$
$JETTIES_0$	$\stackrel{def}{=} (jRel_1.JETTIES_1)$
$MODEL$	$\stackrel{def}{=} (TUGS_2 \mid JETTIES_2 \mid BOAT \mid BOAT \mid BOAT) \setminus \mathbf{L}(MODEL)$

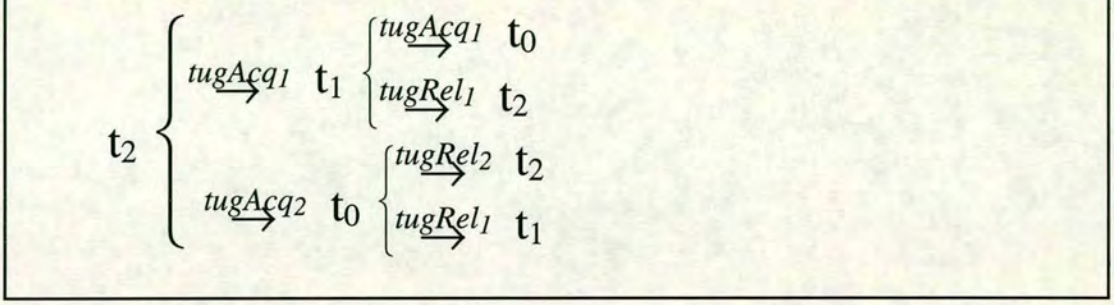
This model simplifies the original model by allowing only three boats. That is sufficient, since it produces the deadlock. A simple proof then shows that this result generalises to larger numbers of boats. In other models it might be necessary to have more instances, depending on the number of interlocking resource acquisitions involved. The question of how many instances of each process type may be needed is examined in more detail below. Now there is a simple enough model to analyse by hand. The following transition diagrams are shown in Figure 6.23.

Figure 6.23: Transition diagram for deadlocking harbour model

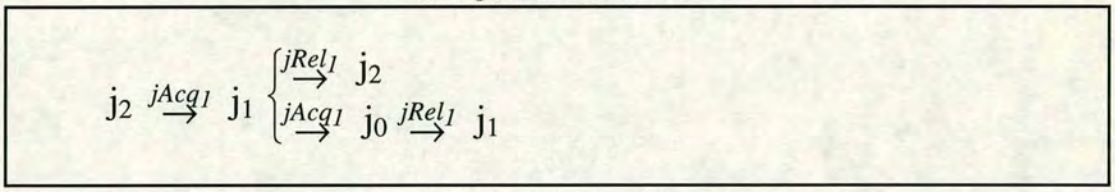
a: a boat



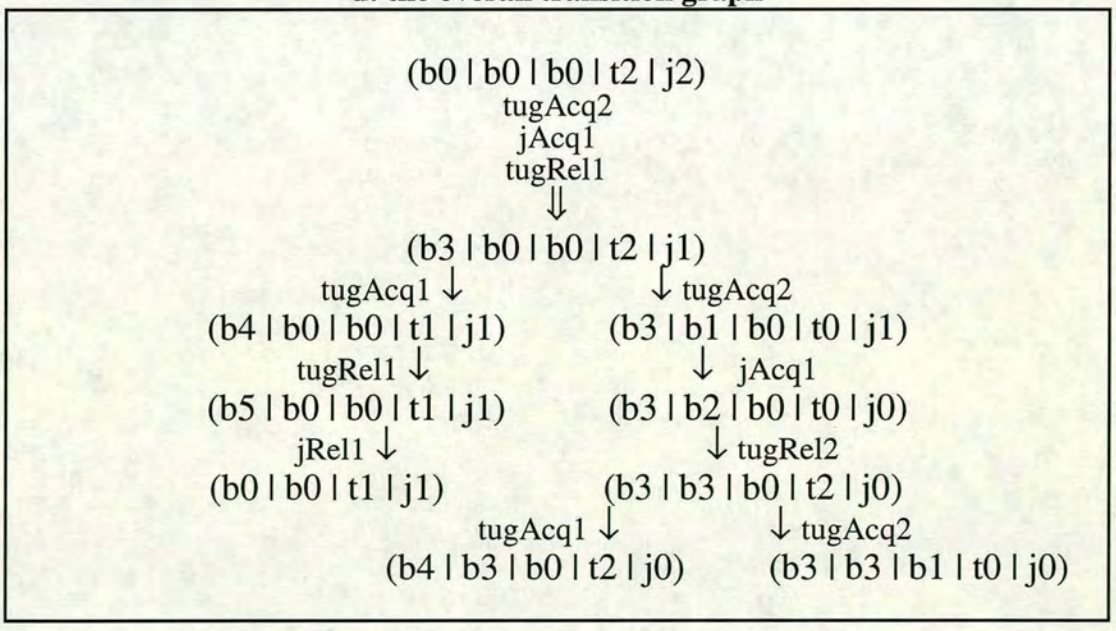
b: a Tugs resource



c: a Jetties resource.



d: the overall transition graph



Note in order to simplify the proof that follows, that any acquisition of a resource creates an agent capable of accepting its release and that this effect is cumulative over acquisitions. This means that releases are never capable of blocking the actions

of an agent. Thus it is safe to assume that a *BOAT* will never be blocked once it has reached b4. In analysing the overall model's transition diagram we can take advantage of this to ignore paths reaching a combination containing this point, since the corresponding boat will be guaranteed to be able to complete and so leave at most two others, which can easily be shown to be deadlock free for the amounts of resource specified. Using these building blocks produces the overall state transition diagram for the model. The state (b3 | b3 | b1 | t0 | j0) is a deadlock. This shows that this model is capable, under certain timings or choices of action, of deadlocking.

A Concurrency Workbench experiment

With these insights a concurrency Workbench experiment was conducted, which demonstrated the expected behaviour. The full experiment is given in Appendix C. In figure 6.24, only part of the output from the Workbench's `fdobs` command is given, showing just the sequence of states leading to deadlock. The deadlock state is marked with a double asterisk.[1] Thus, the workbench agrees with the expected behaviour.

Figure 6.24: Concurrency workbench experiment

a: CCS model for the Concurrency Workbench

```
bi B1 'ja1.B2
bi B2 'tr2.B3
bi B3 'ta1.B4
bi B4 'tr1.B5
bi B5 'jr1.0

bi Tugs2 (ta1.Tugs1)+(ta2.Tugs0)
bi Tugs1 (ta1.Tugs0)+(tr1.Tugs2)
bi Tugs0 (tr1.Tugs1)+(tr2.Tugs2)

bi Jetty2 (ja1.Jetty1)
bi Jetty1 (ja1.Jetty0) + (jr1.Jetty2)
bi Jetty0 (jr1.Jetty1)

bi Model (Tugs2 | Jetty2 | B0 | B0 |
B0)\{ta1,ta2,tr1,tr2,ja1,jr1}
```

[1] The concurrency workbench regards any state where no further actions are possible as a deadlock. Thus it shows two deadlock states in the complete output, the true deadlock and the state where all processes have reached **0**, the CCS passive state.

b: Selected results from `fdobs` command

```
===  ===> Model      *
===  ===> (Tugs0 | Jetty2 | B0 | B0 | B1)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty1 | B0 | B0 | B2)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs2 | Jetty1 | B0 | B0 | B3)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty1 | B0 | B1 | B3)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty0 | B0 | B2 | B3)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs2 | Jetty0 | B0 | B3 | B3)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty0 | B1 | B3 | B3)\{ja1,jr1,ta1,ta2,tr1,tr2}**
```

Generalising the result to larger numbers of boats

It is straightforward to prove that a model which has the potential to deadlock with n processes of a certain type retains this potential with n+1 processes, so long as one of the processes can proceed to termination on its own and releases all the resources it has used in doing so. If it does so, the model reduces to its equivalent with one less process of this type. Since what remains is known to potentially deadlock, the original model could do so under the correct choices or timings. Thus for open models, i.e. models where certain process types are both generated and terminate, a proof of potential deadlock for n of one of these process types is a proof for the same model with n+1.

Probability of deadlock in the model

The probability of deadlock in such a model can be seen to be the probability of it choosing any of the paths leading to a deadlock state. Since this can only happen if a minimum number of processes of each type is present concurrently, the probability of deadlock has an upper bound given by the probability that this number of processes is reached. What is more, in the harbour model the deadlock state occurs only when two boats are tied up unloading and a third one acquires two tugs. Thus the probability of deadlock is the probability of this transition happening conditioned on the probability of two boats being tied up. This informal reasoning about probabilities would require considerable further work to produce a general approach to posing questions about stochastic models, but it is interesting to speculate how the modal μ-calculus or a similar logic might be used in this way.

Comparison with the DEMOS model

To return to the original DEMOS model of Birtwistle, it is necessary to extend the *TUGS* agent to include a $TUGS_3$ state and re-introduce the corresponding actions.

This gives a simplified 3 tug resource and new transition graph shown in Figure 6.25.

Figure 6.25: CCS and transition graph changes for three tug harbour

a: CCS model

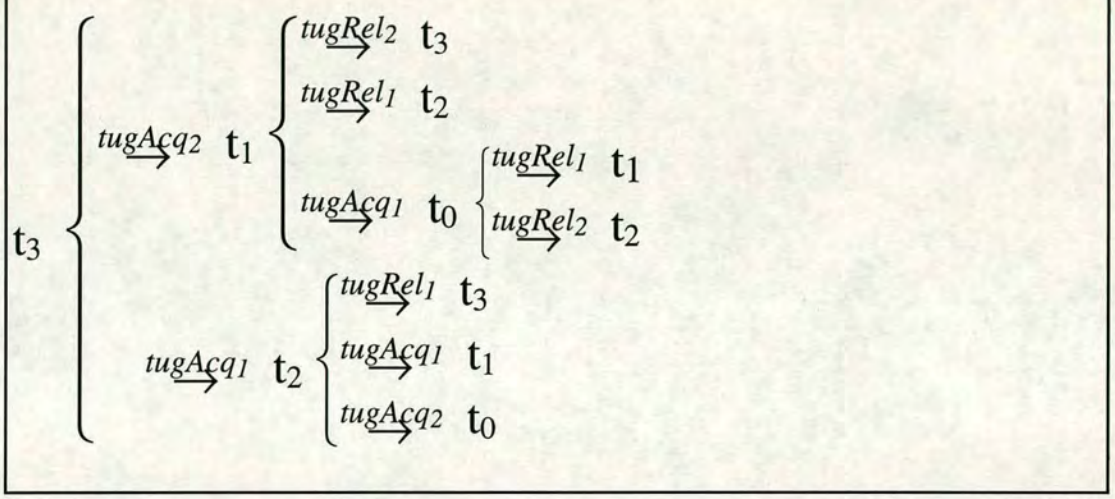
$$TUGS_3 \stackrel{\text{def}}{=} (tugAcq_1.TUGS_2) + (tugAcq_2.TUGS_1)$$

$$TUGS_2 \stackrel{\text{def}}{=} (tugAcq_1.TUGS_1) + (tugAcq_2.TUGS_0) + (tugRel_1.TUGS_3)$$

$$TUGS_1 \stackrel{\text{def}}{=} (tugAcq_1.TUGS_0) + (tugRel_1.TUGS_2) + (tugRel_2.TUGS_3)$$

$$TUGS_0 \stackrel{\text{def}}{=} (tugRel_1.TUGS_1) + (tugRel_2.TUGS_2)$$

b: transition graph



In a similar way to the two tug model overall state transition diagram can be generated and examined for deadlock states. Since the resulting graph is rather complex, it is given in Appendix C. It shows no such states. This is reasonable, since the maximum number of un-terminated boats that can ever be past b0 at any one time is bounded by:

the number of jetties available (2), which limits the number at stages b2..b5;

the number of boats that have acquired the tugs they need to be at stage b1, b4 or b5;

the fact that boats at b4 or beyond are bound to terminate and so may be discounted.

Taking these facts together, the worst case, of boats beyond b0 and not guaranteed termination, is two boats at b3 (unloading) and one at b1 (waiting for one of the b3 boats to leave). Only if the b3 boats are both blocked as a result can deadlock occur. In the two tug model, this worst case led to deadlock as no tugs or jetties were then free. However, in the three tug model, at most one boat can be in state b1 and if this is so at least one boat can always leave state b3 and so terminate, freeing a jetty and a tug. Thus the three tug model is guaranteed deadlock free. In fact no model with an odd number of tugs can ever deadlock.

This leads to the conclusion that the DEMOS model must be incorrect or that the DEMOS solver executes it incorrectly in terms of the CCS definition of its semantics, since it demonstrably does deadlock. In fact, recalling that unmodified DEMOS defines Acquire as always operating on a first come first served basis, some processes, requiring smaller numbers of a resource but arriving later, are thereby blocked unnecessarily. This ensures that many starvation conditions cannot arise, but introduces more cases of seeming deadlock.

Testing with the Concurrency Workbench

Figure 6.26: Testing three tug model with Concurrency Workbench

a: CCS model

```
bi B0 'ta2.B1
bi B1 'ja1.B2
bi B2 'tr2.B3
bi B3 'ta1.B4
bi B4 'tr1.B5
bi B5 'jr1.0

bi Tugs3 (ta1.Tugs2)+(ta2.Tugs1)
bi Tugs2 (ta1.Tugs1)+(ta2.Tugs0)+(tr1.Tugs3)
bi Tugs1 (ta1.Tugs0)+(tr1.Tugs2)+(tr2.Tugs3)
bi Tugs0 (tr1.Tugs1)+(tr2.Tugs2)

bi Jetty2 (ja1.Jetty1)
bi Jetty1 (ja1.Jetty0) + (jr1.Jetty2)
bi Jetty0 (jr1.Jetty1)

bi Model (Tugs3 | Jetty2 | B0 | B0 | B0)\{ta1,ta2,tr1,tr2,ja1,jr1}
```

b: Selected output

```
===  ===> Model
===  ===> (Tugs1 | Jetty2 | B0 | B0 | B1)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty1 | B0 | B0 | B2)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs3 | Jetty1 | B0 | B0 | B3)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty1 | B0 | B1 | B3)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty0 | B0 | B2 | B3)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty0 | B1 | B3 | B3)\{ja1,jr1,ta1,ta2,tr1,tr2}*
===  ===> (Tugs0 | Jetty0 | B1 | B3 | B4)\{ja1,jr1,ta1,ta2,tr1,tr2}
```

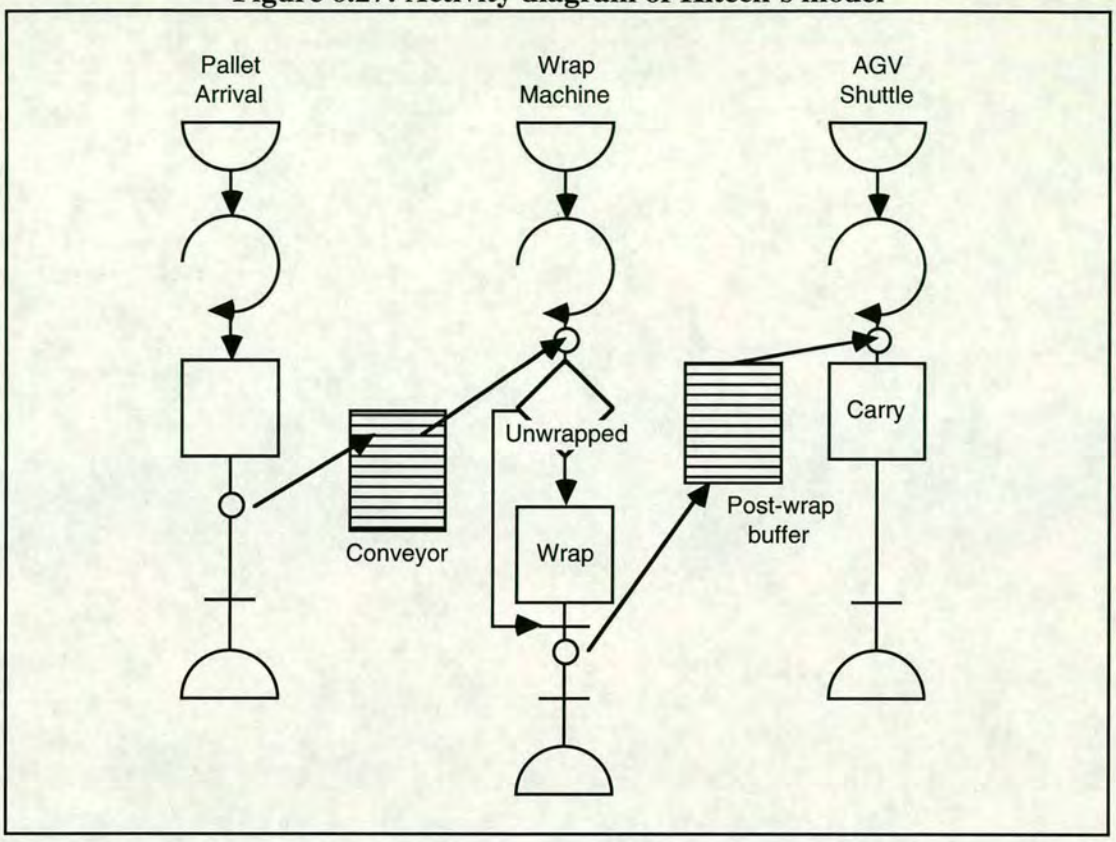
Again the key states have been selected. The previous deadlock state, marked with an asterisk, is passed to reach a path to completion.

6.3.4 Backward propagation of blocking

An extremely common problem in analysing the effects of a simulation model is establishing where the root of a phenomenon lies. This is most often due to *backward propagation* of a problem due to blocking. Thus a slow process emptying a finite buffer may cause a process which is filling that buffer to appear too slow. To consider whether CCS can help us to analyse this sort of problem, consider a small case study reported by a consultant.

Kiteck [49] reported on the use of a discrete event simulation package with animated output of state changes in a warehouse simulation. An extended activity diagram of the core of this model is shown in Figure 6.27. The execution of the model revealed that a Wrap Machine was unable to empty its input conveyor belt fast enough to keep up with the incoming stream of pallets. The graphical animation showed very clearly pallets clogging the conveyor and led to the, erroneous, initial conclusion that the wrap machine was too slow.

Figure 6.27: Activity diagram of Kiteck's model



Closer analysis of the arrivals showed that, in fact, the Wrap Machine did not have to do anything with most of the pallets and was merely acting as buffer space for them. The true problem was that there was a one place buffer beyond the Wrap Machine, where pallets waited for an automatic guided vehicle (AGV) shuttle to carry them into the warehouse. Since it was not always able to remove pallets fast enough, the wrap machine often sat idle, acting as a passive buffer, rather than getting on with its job of wrapping. The problem was propagated backwards and the use of animation obscured the true cause of the problem.

In essence the question that needed answering was, "If the Wrap stage appears blocked, is there some later stage which could be causing this?" More formally, it is necessary to determine whether the inability of the Wrap Machine to perform its input action might be due to an output action being unable to proceed and, if so for which process that action was waiting. This must then be repeated for the blocking process and so on, until no further blocking can be identified.

The CCS model for Kiteck's warehouse is straightforward and is given in outline in Figure 6.28. In this un-timed version no distinction is made between pallets needing wrapping and those not. An alternative model is given in appendix C, showing how a mixture of the two sorts could be modelled.

Figure 6.28: CCS version of Kiteck's model

$Arrival$	$\stackrel{def}{=}$	$\overline{cbAdd_1}.Arrival$	
$WrapMC$	$\stackrel{def}{=}$	$\overline{cbRem_1}.Wrapping$	
$Wrapping$	$\stackrel{def}{=}$	$\overline{oBuffAdd_1}.WrapMC$	
$AGVShuttle$	$\stackrel{def}{=}$	$\overline{oBuffRem_1}.AGVShuttle$	
$OBuff_1$	$\stackrel{def}{=}$	$oBuffRem_1.OBuff_0$	
$OBuff_0$	$\stackrel{def}{=}$	$oBufAdd_1.OBuff_1$	
$CBelt_0$	$\stackrel{def}{=}$	$cbAdd_1.CBelt_1$	
$CBelt_n$	$\stackrel{def}{=}$	$cbAdd_1.CBelt_{n+1} + cbRem_1.CBelt_{n-1}$	$0<n<\text{Limit}$
$CBelt_{\text{Limit}}$	$\stackrel{def}{=}$	$cbRem_1.CBelt_{\text{Limit-1}}$	
$Model$	$\stackrel{def}{=}$	$(Arrival \mid WrapMC \mid AGVShuttle \mid OBuff_1 \mid CBelt_0)\backslash L(Model)$	

It is simple to discover potential blocking in most cases, by removing the process under consideration from the model and seeing which other processes become blocked. Thus, if the AGV Shuttle is removed, it is clearly possible for the Wrap Machine to be unable to proceed once the Post-wrap Buffer is full. This is unsurprising, even from the original extended activity diagram, in such a simple case, but might not be obvious from more complex models. Thus it is possible to claim a double benefit of the approach being developed. Firstly, the clarity of the graphical notation may help in the identification of possible problems. Secondly, the ability to perform rigorous examinations of questions, allows their unambiguous resolution.

There is a possibility of using the modal μ-calculus for examining this sort of question, but it seems to require that the blocked state be fully developed. Again this may be due to lack of full understanding of the possibilities and should remain an open issue.

6.4 Using hierarchies and sub-models

Most of the basic concepts of hierarchy in process based simulation were discussed in chapter 4 and in section 6.2. In order to make effective use of sub-models in a formally defined simulation language, it is necessary either to store the CCS model along with the *modified* DEMOS one or to be able to generate it at need. In fact the most economical way of storing libraries of sub-models would be as the internal representation of an Extended Activity Diagram (DIA format), from which all versions can more or less be generated automatically.

The potential for problems resulting from the use of externally defined sub-models can be illustrated by returning to the example developed in Figure 6.10. There it was shown that simplification of the Stream process would not modify the functional behaviour of the overall model. This might seem an innocuous example, but by a simple change to the interpretation of this process, a very dangerous change is produced. If the Memory resource is not made local to the Stream process, the effect of introducing it into a model is totally different, since it could then compete for this resource with other processes. Fortunately this sort of thing, which may be ambiguous in the sort of activity diagram used in Figure 6.10, is very clear in the CCS model, since the actions to acquire and release the resource are no longer restricted to the Stream process. As with many of the problems discussed here, this seems obvious once it is pointed out, but in the context of re-using predefined sub-models it represents a very real danger.

6.5 Conclusions

This chapter has examined the usefulness of the ideas developed earlier in this dissertation in the light of a number of problems and examples. It has attempted to demonstrate that the basic thesis, that it is *possible* to formalise process based discrete event simulation models in terms of CCS, should be strengthened to say that it is *useful* to simulation modellers to be able to do so.

A number of issues have been raised or left unresolved by this chapter, which suggest that there is scope for further work on this topic.

6.5.1 Successes using CCS

From the examples considered in this chapter, it seems reasonable to claim that the use of CCS is a clear aid to a simulation modeller. The presence of a formal alternative to the activity diagram and DEMOS representations offers another view of a model. Since it is structured and testable, many simple errors can be eliminated. In particular, interfaces and hiding assumptions can be examined, preventing accidental scope errors and highlighting the effects of sub-model combination. Assumptions about independence and concurrency can be carefully checked.

6.5.2 Successes with the modal μ-calculus

Deadlock detection and potential starvation checking with the aid of the modal μ-calculus are made possible. This provides an important step in verifying models. Other questions can be posed for specific models.

6.5.3 Failures using CCS

The interleaving semantics of CCS and TCCS, while built on similar assumptions to the execution of discrete event simulation models on single processor machines, cannot deal with either stochastic or continuous states. This is a limit to their precision when dealing with models, since the results are inevitably conservative with respect to a model operating under specific assumptions. The need to express if-then-else as a guard on one branch and a guard testing the complement on the other often leads to unwieldy summations to test the complement of a single integer. Tests of this sort are impractical and force the use of simplified CCS models with reduced ranges when applying the Concurrency Workbench.

Attempts to incorporate time into these models were frustrating. The TCCS view of time is inadequate to express the really interesting problems and explicit synchronisation actions were often necessary to force actions to occur before time advanced. It seems that a review of alternative forms of time advance and synchronisation is needed to improve on this.

6.5.4 Failures with the modal μ-calculus

Identification of redundant states with CCS is not helped by the calculus, since it does not deal in unreachable states. The learning curve for the calculus is very steep and makes it unlikely that modellers will choose to use it. It does not offer a means to ask probabilistic questions at the moment and much further work is necessary to

develop an equivalently powerful level of interrogation of simulation models. It was of limited general use for questions of genuine concurrency and of blocking propagation. It cannot deal effectively with general questions of fairness.

This area was tackled very late in the work described and should be the subject of further, careful research.

6.5.5 Limits of the Concurrency Workbench

The Concurrency Workbench, in the form available when the work of this dissertation was carried out, proved useful, but unwieldy and less helpful than might have been the case. It is also limited by the rapid increase in the memory demands it makes as the number of states in a model increases. Checking for redundant states is quite cumbersome, even with the help of the Workbench.

The lack of a fully integrated value passing syntax is a serious omission in the current version. A converter from value passing to basic calculus and this should be included in the working system. The lack of an if-else construct in the Workbench syntax is particularly regrettable.

More thought needs to be given to ease of understanding when presenting output. Commands such as `min` and `fdobs` would be much easier to use if their output was related more closely to the structure of the models on which they operate. It appears that some of these issues are being addressed in version 7 of the Workbench, but this was not available at the time of this work.

6.5.4 Further work

This chapter has shown that the application of process algebra techniques provides both an aid to writing good simulation models and a complement to them in terms of the range of questions that can be answered. The major challenges are in closer coupling of the two approaches. In particular the following areas seem obvious targets for further research:

integration of the tools involved to allow Concurrency Workbench functions to be available more directly and in forms more closely related to the activity diagram description of models;

development of new process algebra querying functions targeted at the questions asked by simulation modellers, such as redundant state detection and concurrent action analysis;

extension to stochastic and real valued models, possibly using PEPA or a similar algebra;

development of an applicable process logic for use with the above, expressed at an appropriate level for simulation modellers.

Chapter 7

Conclusions

7.1 General

The results of this dissertation can be summarised as follows.

1. The semantics of discrete event simulation languages are at present poorly described, but can be investigated using a process algebra, such as CCS to formalise the description of the interactions involved.

2. It is possible to design a graphical formalism which is sufficiently powerful to express the most widely used features of process oriented discrete event simulation and to generate executable models directly from these when suitably annotated.

3. It is possible to extend this to express the hierarchical construction and decomposition of such models and to generate executable models and re-usable component sub-models from these diagrams, when suitably annotated.

4. Using a suitably revised version of a discrete event simulation language, in this case DEMOS, it is possible to show important properties such as equivalence, liveness and starvation without resorting to execution of the models, by means of analysis of an equivalent process algebra model.

5. A tool based on a graphical interface can be constructed to support automatic generation of both executable simulation models and equivalent process algebra models.

7.2 Semantics of discrete event simulation

Existing simulation languages are defined informally and precise definitions are often

buried in manuals. Since discrete event simulation usually proceeds on a monoprocessor system, some sort of event interleaving is always required and genuinely simultaneous events have to be scheduled in a deterministic order to allow reproducibility of results.

Chapter 3 demonstrated that the use of a process algebra, such as CCS, helps the designer of such a language avoid ambiguities. It can also allow users of the language to test their understanding of the semantics of constructs in the language.

7.3 Deciding properties of discrete event models

The difficulty of knowing whether a model accurately represents the behaviour of the system it is intended to model is central to the credibility of discrete event simulation. Since any given run of a model is a random walk throught the event space of the model, execution is not an adequate means of establishing such behaviourial properties. It has been shown, however, that the use of CCS to represent a model can allow us to analyse some important properties in advance of quantitative simulation.

Furthermore, the preservation of important properties under simplification and restructuring is important when trying to formulate an efficiently solvable model. Again, the use of CCS has been shown to be of help in locating and exploiting simplifications and component based re-use of sub-models.

7.4 Automating the analysis of simulation models

While many tools for graphical generation of simulation models have appeared in the last few years, none have any formal underpinning. It has been shown here that such a tool can be built to incorporate both discrete event simulation and process algebra behaviourial analysis from a common representation.

Furthermore, the notion of hierarchical modelling has been developed in terms of the graphical, simulation and process algebra representation of models. This greatly aids the use of graphical techniques for large and complex models. The system developed here is likely to be very useful in situations where large models are being built and component models are being re-used from libraries.

7.5 Further work

The work of this dissertation has been developed in terms of the basic Calculus of

Communicating Systems. While this has been shown to be powerful, general and useful, it does not allow for the use of additional information available to the modeller. In particular, it does not deal with stochastic measures of time and non-determinism. Nor does it have useful ways of handling sequential execution of moedls of concurrent systems. The alternatives of stochastic and synchronous process algebras need to be explored in some detail to assess their contribution.

Closer integration of the graphical interface and the functional analysis software is very desirable. Even if the Concurrency Workbench could be tailored to ask the questions requiring answers in simulation modelling, it presents its results in an opaque manner. Reimplementation of these features within the context of Activity Diagrams seems a fruitful approach. At the same time some new forms of analysis might be added, to deal in more specific ways with questions such as concurrent event modelling. The modal μ-calculus is too obscure to be used directly.

Finally a considerable amount of work remains in evaluating the approaches developed here on real modelling problems. This is much more possible as a result of the tools developed in this work.

7.6 Assessment

The results of this work are not all positive, but in a number of areas the usefulness of combining functional analysis of systems with simulation modelling has been clearly shown. The avenues still open suggest that some at least of the open issues can be resolved. The overall assessment is, therefore, that the work reported has been worthwhile and has potential for exploitation.

References

1. Agerwala T. 1979. "Putting Petri Nets to Work", IEEE Computer, December, 85-94.

2. Ajmone Marsan M, G. Conte and G. Balbo..1984. "A Class of Generalised Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems", ACM Transactions on Computer Systems, 2 (2) (May) 93-122.

3. Aldwinckle John, Rajagopal Nagarajan and Graham Birtwistle 1992 "An Introduction to Modal Logic and its Applications on the Concurrency Workbench", University of Calgary, Department of Computer Science Technical Report

4. Auyong L.S. 1991. *A Graphical Editor and Checker for Stochastic Petri Nets*, MSc Dissertation, University of Edinburgh, Department of Computer Science.

5. Auyong L. S. and R. Pooley 1992 "An Editing and Checking Tool for Stochastic Petri Nets - esp", in J. Stephenson Ed. *Proceedings of the European Simulation Multi-conference, York,* July 1992, SCS Europe

6. Barber E.O. and P.H. Hughes August 1990, "Evolution of the Process Interaction Tool, A Graphical Editor for DEMOS", in *Proceedings of the Seventeenth SIMULA Users Conference, Pilsen*, pp 171-180, Association of SIMULA Users

7. Bauman R. and T.A. Turano November 1986, "Production language simulation of Petri nets", Simulation Vol 47 No 5, pp 191-198

8. Beilner H. and F.J. Stewing 1987. "Concepts and Techniques of the Performance Modelling Tool HIT", in *Proceedings of the European Simulation Multiconference*, Vienna, SCS Europe

9. Beilner H. June 1989. "Structured Modelling - Hierarchical Modelling", in Proceedings of the European Simulation Multiconference, Rome, SCS International

10. Beilner H., J. Mäter and C. Wysocki 1994. "The Hierarchical Evaluation Tool HIT", in *Proceedings of the 7th International Conference on Techniques and Tools for Computer performance Evaluation*, Vienna pp 3-6

11. Belsnes D. and K.A. Bringsrud May 1978. "X.25 Implemented in SIMULA", in *Eurocomp 78*, London

12. Birtwistle G.M., O.J.Dahl, B.Myhrhaug and K.Nygaard 1973 *SIMULA BEGIN*, Studentlitteratur, Lund.

13. Birtwistle G.M. 1979. *Discrete Event Modelling On SIMULA*, Macmillan, London

14. Birtwistle G.M. and P. Luker February 1984. "Dialogs for Simulation", in Birtwistle and Luker Ed. *Proceedings of the conference on Simulation in Strongly Typed Languages*, San Diego, pp90-95, Society for Computer Simulation, La Jolla, California

15. Birtwistle G.M., P. Luker, G. Lomow and B. Unger 1985. "Process Style Packages for Discrete Event Modelling: Experience from the Transaction, Activity and Event Approaches", Transactions of the SCS 2(1), pp25-56

16. Birtwistle G.M., C. Tofts and R.J. Pooley October 1993, "Characterising the Structure of Simulation Models in CCS", Transactions of the SCS Vol 10 No 3, pp 205-237

17. Buchholz P. March 1989 "Definitions of Submodels and Classification of Aggregates, ESPRIT II IMSE Project Report, Universität Dortmund

18. Chiola G. March 1987. "A Graphical Petri Net Tool for Performance Analysis", in D. Potier Ed. *Proceedings of the International Workshop on Modelling Techniques and Performance Evaluation*, pp 297-307, AFCET, Paris

19. Chiola G., C. Dutheillet, G. Franceschinis and S. Haddad 1991 "On Well Formed Coloured Nets and their Symbolic Reachability Graph", in K. Jensen and G. Rozenburg, editors, *High Level Petri Nets: Theory and Application*, Springer Verlag

20. Cleaveland R., J. Parrow and B. Steffen "The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems", ACM TOPLAS, Vol 15 No 1, 1993

21. Clementson A. 1973. *Extended Control and Simulation Language Manual*, Birmingham University, England

22. Cota B.A. and R.G. Sargent 1992 "A Modification of the Process Interaction World View", ACM Transactions on Modelling and Computer Simulation, Vol 2 No 2, April 1992, pp 109-129

23. Dahl O.J. and K. Nygaard 1970 *Simula 67 Common Base Language*, Norwegian Computing Centre, Oslo, Norway, 2nd Edition.

24. Dam Mads June 1992 *CTL* and ECTL* as fragments of the modal m-calculus*, Edinburgh University, Dept of Computer Science, Report ECS-LFCS-92-217.

25. Dumas M.B. August 1984 "Simulation Modeling for Hospital Bed Planning", Simulation Vol 43 No 2, pp 69-78

26. Evans J.B. 1988, *Structures of Discrete Event Simulation*, Ellis Horwood, Chichester

27. Franta W.R. 1978. *The Process View of Simulation*, North-Holland, Amsterdam

28. Fayek A.M., D. van Welden and G. Vansteenkiste 1990 "Applying DEVS Methodology to Continuous Systems Modelling", Simulation Digest, Vol 21 No 1, pp 39-45, ACM SIGSIM/IEEE TC SIM

29. Garcia M.R. 1990 "Discrete Event Simulation Methododlogies and Formalisms", Simulation Digest, Vol 21 No 1, pp 3-13, ACM SIGSIM/IEEE TC SIM

30. Gilmore S. and J. Hillston May 1994. "The PEPA Workbench: A Tool to Support a Process Algebra-Based Approach to Performance Modelling", in Günter Haring and Gabriele Kotsis (Eds) *Computer Performance Evaluation, Modelling Techniques and Tools*, Springer-Verlag LNCS-794, pp353-368

31. N. Götz, U. Herzog and M. Rettelbach "Multiprocessor and Distributed System Design: the Integration of Functional Specification and Performance Analysis using Stochastic Process Algebras", in *Proceedings of Performance '93*, 1993

32. Hennessy, M.C. and A.J.R.G. Milner 1985. "Algebraic Laews for Non-determinism and Concurrency", Journal of ACM, Vol 32 No 1, pp137-161

33. Hills P.R. 1965 "SIMON a Simulation Language in Algol", in S.M. hollingdale Ed. *Simulation in OR*, English Universities Press, London

34. Hills P.R. 1966. *Hand or Computer Simulation System (Hocus)*, Management Engineering Report 66/4, Imperial College, University of London, England

35. Hills P.R. 1968. "Hocus a simple approach to simulation", Data Processing, May 1968

36. Hillston J. September 1992. "A Tool to Enhance Model Exploitation", in R.J. Pooley and J.E. Hillston Eds. *Computer Performance Evaluation, Modelling Techniques and Tools*, Edinburgh University Press Edits 10, pp131-142

37. Hillston J.E. April 1994. *A Compositional Approach to Performance Modelling*, PhD Dissertation CST-107-94, University of Edinburgh, Department of Computer Science

38. Hoare, C.A.R. 1985. *Communicating Sequential Processes*, Prentice Hall, London.

39. Hoover S.V. and R.F. Perry 1989 *Simulation: A Problem Solving Approach*, Addison-Wesley

40. Hughes P.H. 1984. *DEMOS activity Diagrams*, Notat nr 1, FAG 45080 Simulering, Høst 1984, Norges Tekniske Høgskole, Institutt for Databehandling, 7034 Trondheim-NTH, Norway

41. Hughes P.H., R.J. Pooley, T.K. Rylance and J.F. Smith June 1986. *Simmer Pilot Capability*, SIMMER Document SAG/001/4, STL, Copthall House, Newcastle-under-Lyme, England

42. Hughes P.H. 1986. "The Design of a Performance Modelling Environment", in *Proceedings of the 14th SIMULA Users Conference, Stockholm,* pp123-135, Association of SIMULA Users, Postbox 4403 Torshov, N-0402 Oslo 4, Norway

43. Information Systems Research Associates 1986. *PAWS Users Guide*

44. Information Systems Research Associates 1985. *GPSM Manual*

45. Jang, H. and T.G. Kim 1990 "Performance Modelling of Relational Database Systems on Multicomputers", Simulation Digest, Vol 21 No 1, pp 29-38, ACM SIGSIM/IEEE TC SIM

46. Jankowski P.L. and J.W. Rozenblit 1990 "DEVS-Scheme Simulation of Stream Water Quality", Simulation Digest, Vol 21 No 1, pp 20-28, ACM SIGSIM/IEEE TC SIM

47. Jou C-C and S. Smolka August 1990 "Equivalences, Congruences and Complete Axiomatisations of Probabilistic Processes" in J.C.M. Baeten and J.W. Klop Eds. *CONCUR '90*, Springer LNCS 458, pp 367-383

48. Kim, T.G. and B.P. Zeigler 1990 "The DEVS Scheme Simulation and Modelling Environment" in P.A. Fishwick and R.B. Modejeski Eds. *Knowledge Based Simulation: Methodology and Applications*, Springer Verlag

49. Kiteck P. September 1991 "Analysis of Component Interaction in a Distribution Facility Using Simulation", Proceedings of the First EUROSIM Congress, Capri, EUROSIM

50. Kiviat P.J., R. Villanueva and H.M. Markovitz 1968. *The Simscript II Programming Language*, Prentice Hall, Eaglewood Cliffs, New Jersey

51. Kiviat P.J., R. Villanueva and H.M. Markovitz 1983. *The Simscript II.5 Programming Language* (edited by A. Mularney), CACI, Los Angeles

52. Kurose J.F., J.G. Kurtiss, R.F. Gordon, E.A. McNair and P.D. Welch March 1986. *A graphics-oriented modeller's workstation environment for the RESearch Queueing Package (RESQ)*, IBM Research Report RC11803, IBM Hawthorne Laboratories, Yorktown Heights, NY10596

53. Larsen K. and A. Skou September 1991 "Bisimulation through Probabilistic testing", Information and Computation, 94(1), pp1-28

54. Law A.M. and W.D. Kelton 1991 *Simulation Modelling and Analysis*, 2nd Ed. McGraw-Hill

55. Lindemann C. September 1992. "DSPNExpress: a Software Package for the Efficient Solution of Deterministic and Stochastic Petri Nets", in R. Pooley and J. Hillston Eds. *Computer Performance Evaluation - Modelling Techniques and Tools*, 6th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Edinburgh, Edits 10, Edinburgh University Press,

56. Mathewson S.C. December 1974. "Simulation program generators", Simulation Vol 23 No 6, pp 181-189

57. Melamed B. and R.J.T. Morris 1985. "Visual Simulation: the Performance Analysis Workstation", IEEE Computer, Vol 18 No 8, pp 87-94, August 1985

58. Milner, R. 1990. *Communication and Concurrency*, Prentice Hall, London.

59. Mitrani, I. 1982. *Simulation Techniques for Discrete Event Systems*, Cambridge Computer Science Texts 14, Cambridge University Press

60. Moller, F. and C. Tofts. 1989. *A Temporal Calculus of Communicating Systems*, Edinburgh University, Department of Computer Science, Report ECS-LFCS-89-104.

61. Moller F. October 1992. *The Edinburgh Concurrency Workbench (Version 6.1)*, Edinburgh University, Department of Computer Science, LFCS Technical Note, TN34

62. Molloy M.K. and P. Riddle April 1986. *The stochastic Petrinet analyser system design tool for bit-mapped workstations*, Technical Report, Department of Computer Science, University of Texas at Austin

63. Molloy M.K. Sept 1982 "Performance Analysis Using Stochastic Petri Nets", IEEE Trans. Computers, C-31 9, pp 913-917

64. Nance R. "The Conical Methodology for Simulation", Technical Report, Virginia Technical University

65. Peck S. 1992 "Modelling Discrete-Event Systems Using Co-opting Processes", in P. Luker Ed. *Proceedings of Summer Computer Simulation Conference, Reno, Nevada*, SCS, La Jolla, California

66. Pegden C.D. 1985 *Introduction to SIMAN*, Systems Modeling Corp., State College, Penn.

67. Pooley R.J. 1979 *A Model of the CCITT X.25 Protocol, Level 3*, MSc dissertation, Computer Science, University of Bradford, October 1979

68. Pooley R.J. August 1984. "You don't have to be big to be beautiful", in *Proceedings of 12th SIMULA Users Conference, Budapest*, pp 69-81, Association of SIMULA Users, Postbox 4403 Torshov, N-0402 Oslo 4, Norway

69. Pooley R.J. and G.M. Birtwistle 1986. "Process based modelling of communications protocols", in S. Schoemaker Ed. *Computer Networks and Simulation III*, pp 81-101, North Holland, Amsterdam

70. Pooley R.J. 25th-26th September 1986. "Graphics and Modelling", in *Proceedings of 2nd Computer and Telecommunications Performance Engineering Workshop*, Department of Computer Science, University of Edinburgh

71. Pooley R.J. and G.M. Birtwistle January 1987. "Design of a flexible, extensible modelling environment", in *Proceedings of conference on Computer Integrated Manufacturing Systems and Robotics*, San Diego, pp 14-17, Society for Computer Simulation, La Jolla, California

72. Pooley R.J. and M.W. Brown June 1988. "Automated modelling with the General Attributed (Directed) Graph Editing Tool - GA(D)GET", Proceedings of the ESM, Nice, pp 410-415

73. Pooley R.J. September 1987. "Towards a standard for discrete event simulation diagrams", Proceedings of 15th SIMULA Users Conference, pp 141-152, Simula a.s., Postbox 4403 Torshov, N-0402 Oslo 4, Norway

74. Pooley R.J. 1987. *An Introduction to programming in SIMULA*, Blackwells

75. Pooley R.J. 1991. "The Integrated Modelling Support Environment", in G. Balbo and G. Serazzi ed. *Proceedings of the 6th International Conference on Techniques and Tools for Computer Performance Prediction*, Torino, February 1991, Springer Verlag, pp 1-16.

76. Pooley, R.J. 1991a. "Towards a Standard for Hierarchical Process Oriented Discrete Event Simulation Diagrams - Part 1, A Comparison of Existing Approaches", Transactions of the Society for Computer Simulation, Vol 8 No 1 (March), 1-20.

77. Pooley, R.J. 1991b. "Towards a Standard for Hierarchical Process Oriented Discrete Event Simulation Diagrams - Part 2, The Suggested Flat Approach", Transactions of the Society for Computer Simulation, Vol 8 No 1 (March), 21-32.

78. Pooley, R.J. 1991c. "Towards a Standard for Hierarchical Process Oriented Discrete Event Simulation Diagrams - Part 3, Aggregation and Hierarchical Modelling", Transactions of the Society for Computer Simulation, Vol 8 No 1 (March), 33-42.

79. Pooley, R.J. 1991d. *Generalised representation, decomposition and aggregation of process interaction models*, IMSE Project Deliverable D4.4-2 Version 1, University of Edinburgh

80. Pooley R.J. 1993c. "Demographer: A Graphical Tool for Combined Simulation and Functional Modelling", in R.Pooley and R. Zobel Eds, *UKSS '93: Proceedings of the First Conference of the UK Simulation Society*, September 1993, pp 91-95

81. Pritsker A.A.B. 1979. *Modelling and analysis using Q-GERT Networks*, Systems Publishing Corporation, Lafayette, Indiana

82. Pritsker A.A.B. 1984. *Introduction to simulation and SLAM II*, John Wiley and Sons, New York

83. Raeder G. August 1985. "A survey of current graphical programming techniques", IEEE Computer, Vol 18 No 8, pp 11-25

84. Russell E.C. 1983. *Simulation with Processes and Resources in Simscript II.5*, CACI Inc., Los Angeles

85. Russell E.C. and J.S. Anino 1983. *A Quick Look at Simscript II.5*, CACI Inc., Los Angeles

86. Sargent R.G. October 1988 "Event-graph modeling for Simulation with an Application to Flexible Manufacturing Systems", Management Science 34(10), pp 1231-1251

87. Schiffner G. and H. Godbersen May 1986. "Function Nets: A comfortable tool for simulating database system architectures", Simulation Vol 46 No 5, pp 201-210

88. Schriber T.J. 1974. *Simulation Using GPSS*, Wiley, New York

89. Schruben L. 1983. "Simulation Modelling with Event Graphs", CACM Vol 26 No 11, November 1983, pp 957-963

90. Schruben L. 1991 *Sigma: A Graphical Simulation System*, The Scientific Press, San Francisco

91. Sevine S. 1990. "DEVS-CLOS: Implementing DEVS Concepts in Common Lisp Object System", Simulation Digest, Vol 21 No 1, pp 14-19, ACM SIGSIM/IEEE TC SIM

92. Smith C.U. 1990. *Performance Engineering of Software Systems*, Addison-Wesley

93. Som T.K. and R.G. Sargent 1989 "A Formal Development of Event Graphs as an Aid to Structured and Efficient Simulation of Programs", ORSA Journal of Computing, i(2), pp 107-125

94. Standridge, C.R. 1985. "Performing Simulation Projects with The Extended Simulation System (TESS)", Simulation Vol 45 No 6 (Dec), 283-291.

95. Stirling C. 1992. *Modal and Temporal Logics for Processes*, Technical Report ECS-LFCS-92-221, Laboratory for the Foundations of Computer Science, Department of Computer Science, University of Edinburgh

96. Strulo B. 1993. *Process Algebra for Discrete Event Simulation*, PhD Dissertation, Imperial College London

97. Tocher K.D. 1963. *The Art of Simulation*, The English Universities Press, London

98. Tofts, C. 1989. *Timing Concurrent Processes*, Report ECS-LFCS-89-104, Edinburgh University, Department of Computer Science.

99. Tofts C.N.M. December 1992 "Describing Social Insect Behaviour Using Process Algebra', Transactions of the Society for Computer Simulation, 9(4), pp 227-283

100. Tofts C. 1993. *Process Semantics for Simulation*, Technical Report, Department of Mathematics and Computer Science, University of Swansea

101. Törn A.A. 1985 "Simulation nets, a simulation modeling and validation tool", Simulation Vol 45 No 2, August 1985, pp 71-75

102. Trivedi K.S.; G. Ciardo and J.G. Muppala 1991. *Manual for the SPNP Package Version 3.0*, Duke University, Durham.

103. Tumay K. 1987 "Manufacturing Simulation with Simfactory", in *Proceedings of conference on Computer Integrated Manufacturing Systems and Robotics*, San Diego, January 1987 pp 36-38, Society for Computer Simulation, La Jolla, California

104. Vaucher J. 1971 "Simulation Data Structures using SIMULA 67", in *Proceedings of the Winter Simulation Conference*, pp 255-260

105. Vaucher J. 1973 "A Generalised Wait-Until Algorithm for General Purpose Simulation Languages", Proceedings of the Winter Simulation Conference, pp 177-183

106. M. Veran and D. Potier "QNAP 2: a Portable Environment for Queueing System Modelling" in D. Potier Ed. Proceedings of Modelling Techniques and Tools for Computer Performance Evaluation, North Holland, 1985, pp 25-63

107. Walker D.J. 1987. *Introduction to a Calculus of Communicating Systems*, Report ECS-LFCS-87-22, Department of Computer Science, University of Edinburgh

108. Williams A.N. 1979 *A Model of the CCITT X.25 Protocol, Level 2*, MSc dissertation, Computer Science, University of Bradford, October 1979

109. Yücesan E. and L. Schruben 1992 "Structural an Behavioural Equivalence of Simulation Models", ACM Transactions on Modelling and Computer Simulation, Vol 2 No 1, January 1992, pp 82-103

110. Zeigler B.P. 1976 *Theory of Modeling and Simulation*, Wiley, New York

111. Zeigler B.P. 1984 *Multi-facetted Modeling and Discrete Event Simulation*, Academic Press, New York

112. Zeigler B.P. 1990 *Object Oriented Simulation with Hierarchical Modular Models*, Academic Press, New York

Appendix A

Source of Demographer

This Appendix contains the source of the PC version of Demographer used in Cahpter 5 of this dissertation.

```
! DOS graphical editor for DEMOS;

begin
  short integer grm;

  integer Left = 127, Middle = 26, Right = 31; ! Delete,End,PageDn;

  integer MoveL = 5, MoveR = 4, MoveU = 28, MoveD = 14; ! Cursor keys;

! Node types are defined here as ...;

  integer Hold_Sym = 1, Start_Sym = 2, End_Sym = 3, Decision_Sym = 4,
          Synch_Sym = 5, Link_Sym  = 6, Res_Sym = 7, While_Sym = 8,
          Bin_Sym = 9, Store_Sym = 10, Sub_Sym = 11, Max_Sym = Sub_Sym;

! Link symbols indicating direction are ...;

  integer LR  = -1, RL  = -2, DU  = -3, UD  = -4,
          RU  = -5, RD  = -6, LU  = -7, LD  = -8,
           UR  = -9, DR  =-10, UL  =-11, DL  =-12, Del = -13;

! Powers of two used to store current links in Diag table;

  integer L_R = -1, R_L = -2, D_U = -4, U_D = -8,
          R_U =-16, R_D =-32, L_U =-64, L_D =-128,
          U_R =-256,D_R =-512,U_L=-1024,D_L=-2048;

! Direction of current move when linking is one of ...;

  integer Up = 1, Dn = -1, Lf = 2, Rt = -2, NW = 0;

! Colours used are ...;

  integer Black = 0, White = 15, Red = 4, Blue = 1, LGrey = 7, Green = 2,
          Yellow = 14, LGreen = 10;

! Size of grid in squares;

  integer XSquares = 25, YSquares = 15;

  integer button, x, y, ox, oy, ob, d, CurrSymb, PrevSymb, I, J;
  character Current_Char, Prev_Char;
  Boolean Exited, No_Move, Linking, First_Link;
  text String, F_Name;
  text array Titles(DL :Max_Sym,1:6);
  text array Form(0:XSquares,0:YSquares,1:6,1:2);
  integer array Diag(0:XSquares,0:YSquares);
  integer array Lnk(Rt:Lf,DL:0);
  integer array SymbMap(DL:LR);

  ref(InFile) InF;
  ref(outfile) OF;

external class drawing = "..\drawing\drawing";

drawing(16)
begin

   ! This is the prefixed block that does the graphics;

  procedure gen;
  begin

     ! DEMOS generating backend for graphical input programs;

   integer Iden = 1;
   integer          Sched = 2, Successor = 3, Locals = 4; ! Start node;
   integer          Amount = 2;                           ! Res or Bin;
   integer          Period = 2;                           ! Hold;
   integer          Condition =2;                         ! Condition;

   integer Downwards = 1, Upwards = 2, Leftwards = 3, Rightwards = 4;
   ref(outfile) model;
```

```
text F_Name;

   SetColour(LGreen);
   FillSquare(260,0,85,500);
   J := TextLine(280);
SetPos(J,1);
OutText("Name of DEMOS output file");
F_Name :- InText(40).Strip;
model :- new OutFile(F_Name);

inspect model do
begin

procedure OutLine(T,D); text T; integer D;
begin
   ! Print out the text with D spaces of indentation and a newline;
   OutText(Blanks(D)&T);
   OutImage;
end;

text procedure GetNext(T); name T; text T;
begin
   text Res;
   character Ch;
   T :- T.Strip;
   if T==NoText then GetNext :- NoText else
   begin
      Res :- Blanks(80);
      while T.GetChar=' ' do;
      T.SetPos(T.Pos-1);
      Ch := T.GetChar;
      while Ch<>' ' and then T.More do
      begin
         Res.PutChar(Ch);
         Ch := T.GetChar;
      end;
      if not T.More then Res.PutChar(Ch);
      T :- T.Sub(T.Pos,T.Length-T.Pos+1);
      GetNext :- Res.Strip;
   end;
end;

class Global_Item(X,Y,Ident); integer X,Y; text Ident;
begin
   ref(Global_Item) Next;
end;

Global_Item class Dist_Item(Sort, P1, P2); text Sort, P1, P2;
begin
end;

class Global_List;
begin
   ref(Global_Item) First;
   procedure Into(New_Item); ref(Global_Item) New_Item;
   begin
      New_Item.Next :- First;
      First :- New_Item;
   end;
   ref(Global_Item) procedure Get;
   begin
      Get :- First;
      First :- First.Next;
   end;
   Boolean procedure Empty; Empty := First==none;
end;

ref(Global_List) Entity_List, Entity_List2,
                 Res_List, Res_List2,
                 Bin_List, Bin_List2,
                 Store_List, Store_List2,
                 Sub_List, Sub_List2,
                 Dist_List, Dist_List2;

procedure Read_Table;
begin
   integer X, Y, I;
   while not LastItem do
   begin
      X := InInt; Y := InInt;
      Diag(X,Y) := InInt;
       InImage;
       for I := 1 step 1 until 6 do
       begin
          Form(X,Y,I,2) :- Copy(Image.Strip);
          InImage;
       end;
   end;
end;

procedure Prologue;
begin
   outline("begin",0);
```

```
      outline("external class demos;",3);
      outline("DEMOS",3);
      outline("begin",3);
   end;

   procedure Epilogue;
   begin
      OutImage;
      OutLine("Hold(InReal);",6);
      OutLine("end",3);
      OutLine("end",0);
   end;

   procedure Find_Globals;
   begin
      ! Locate all entities, Reses, Bins etc.;
      integer X, Y;
      for X := 0 step 1 until XSquares do
         for Y := 0 step 1 until YSquares do
         begin
            if Diag(X,Y) = Start_Sym then
                Entity_List.Into(new Global_Item(X,Y,Form(X,Y,Iden,2))) else
            if Diag(X,Y) = Res_Sym then
                Res_List.Into(new Global_Item(X,Y,Form(X,Y,Iden,2))) else
            if Diag(X,Y) = Bin_Sym then
                Bin_List.Into(new Global_Item(X,Y,Form(X,Y,Iden,2))) else
            if Diag(X,Y) = Store_Sym then
                Store_List.Into(new Global_Item(X,Y,Form(X,Y,Iden,2))) else
            if Diag(X,Y) = Sub_Sym then
                Sub_List.Into(new Global_Item(X,Y,Form(X,Y,Iden,2)));
         end;
   end;

   procedure Build_Entities;
   begin
      ! Output the class declarations of the entities;
      text T, Dist;
      integer X1,Y1;

      procedure Follow(X, Y, Heading); name X, Y;
                                       integer X, Y, Heading;
      begin
         ! Follow a link to its end;
         switch Coming := Down_W, Up_W, Left_W, Right_W;
         integer Link_Type;

         Link_Type := -Diag(X,Y);
         GoTo Coming(Heading);

Down_W:
         if Link_Type//(512*2)*2<>Link_Type//512 then
         begin
            X := X + 1;
            Heading := Rightwards;
         end else
         if Link_Type//(2048*2)*2<>Link_Type//2048 then
         begin
            X := X - 1;
            Heading := Leftwards;
         end else
         if Link_Type//(8*2)*2<>Link_Type//8 then
         begin
            Y := Y + 1;
            Heading := Downwards;
         end else GoTo Skip;
         GoTo Done;

Up_W:
         if Link_Type//(256*2)*2<>Link_Type//256 then
         begin
            X := X + 1;
            Heading := Rightwards;
         end else
         if Link_Type//(1024*2)*2<>Link_Type//1024 then
         begin
            X := X - 1;
            Heading := Leftwards;
         end else
         if Link_Type//(4*2)*2<>Link_Type//4 then
         begin
            Y := Y - 1;
            Heading := Upwards;
         end else GoTo Skip;
         GoTo Done;

Left_W:
         if Link_Type//(2*2)*2<>Link_Type//2 then
         begin
            X := X - 1;
            Heading := Leftwards;
         end else
         if Link_Type//(128*2)*2<>Link_Type//128 then
         begin
```

```
            Y := Y + 1;
            Heading := Downwards;
         end else
         if Link_Type//(64*2)*2<>Link_Type//64 then
         begin
            Y := Y - 1;
            Heading := Upwards;
         end else GoTo Skip;
         GoTo Done;

Right_W:
         if Link_Type//(1*2)*2<>Link_Type//1 then
         begin
            X := X + 1;
            Heading := Rightwards;
         end else
         if Link_Type//(32*2)*2<>Link_Type//32 then
         begin
            Y := Y + 1;
            Heading := Downwards;
         end else
         if Link_Type//(16*2)*2<>Link_Type//16 then
         begin
            Y := Y - 1;
            Heading := Upwards;
         end else GoTo Skip;
         GoTo Done;

Done:
         if Diag(X,Y)<0 then Follow(X,Y,Heading);
Skip:
      end;

      procedure Follow_Back(X, Y, Heading); name X, Y;
                                            integer X, Y, Heading;
      begin
         ! Follow a link to its origin;
         switch Coming := Down_W, Up_W, Left_W, Right_W;
         integer Link_Type;

         Link_Type := -Diag(X,Y);
         GoTo Coming(Heading);

Down_W:
         if Link_Type//(64*2)*2<>Link_Type//64 then
         begin
            X := X + 1;
            Heading := Rightwards;
         end else
         if Link_Type//(16*2)*2<>Link_Type//16 then
         begin
            X := X - 1;
            Heading := Leftwards;
         end else
         if Link_Type//(4*2)*2<>Link_Type//4 then
         begin
            Y := Y + 1;
            Heading := Downwards;
         end else GoTo Skip;
         GoTo Done;

Up_W:
         if Link_Type//(128*2)*2<>Link_Type//128 then
         begin
            X := X + 1;
            Heading := Rightwards;
         end else
         if Link_Type//(32*2)*2<>Link_Type//32 then
         begin
            X := X - 1;
            Heading := Leftwards;
         end else
         if Link_Type//(8*2)*2<>Link_Type//8 then
         begin
            Y := Y - 1;
            Heading := Upwards;
         end else GoTo Skip;
         GoTo Done;

Left_W:
         if Link_Type//(1*2)*2<>Link_Type//1 then
         begin
            X := X - 1;
            Heading := Leftwards;
         end else
         if Link_Type//(256*2)*2<>Link_Type//256 then
         begin
            Y := Y + 1;
            Heading := Downwards;
         end else
         if Link_Type//(512*2)*2<>Link_Type//512 then
         begin
            Y := Y - 1;
```

```
            Heading := Upwards;
         end else GoTo Skip;
         GoTo Done;

Right_W:
         if Link_Type//(2*2)*2<>Link_Type//2 then
         begin
            X := X + 1;
            Heading := Rightwards;
         end else
         if Link_Type//(1024*2)*2<>Link_Type//1024 then
         begin
            Y := Y + 1;
            Heading := Downwards;
         end else
         if Link_Type//(2048*2)*2<>Link_Type//2048 then
         begin
            Y := Y - 1;
            Heading := Upwards;
         end else GoTo Skip;
         GoTo Done;

Done:
         if Diag(X,Y)<0 then Follow_Back(X,Y,Heading);
Skip:
      end;

      procedure Next_Sym(X,Y); name X,Y; integer X,Y;
      begin
         integer OldX,OldY,W;
         Boolean Failed;
         if Y<>11 then
         begin
            OldX := X; OldY := Y; Y := Y+1;W:=Downwards;
            if Diag(X,Y)<0 then Follow(X,Y,W);
            if X=OldX and Y=OldY+1 and X<>24 then   ! Went nowhere;
            begin
               X := X+1; Y := OldY;W:=Rightwards;
               if Diag(X,Y) <0 then Follow(X,Y,W);
               if X=OldX+1 and Y=OldY and OldX<>0 then   ! Still nowhere;
               begin
                  X := X-2;W:=Leftwards;
                  if Diag(X,Y)<0 then Follow(X,Y,W);
                  if X=OldX-1 and Y=OldY and Y<>0 then   ! Check Upwards;
                  begin
                     X :=OldX; Y := Y-1;W:=Upwards;
                     if Diag(X,Y)<0 then Follow(X,Y,W);
                     Failed :=  X=OldX and Y=OldY-1;
                  end;
               end;
            end;
         end;
         if Failed then
         begin
            Y := Y+1;
            if Diag(X,Y+1)>0 then Y := Y+1 else
            if Diag(X+1,Y)>0 then X := X+1 else
            if Diag(X-1,Y)>0 then X := X-1 else
            if Diag(X,Y-1)>0 then Y := Y-1 else
            begin
               OutLine("***Missing link from***",1);
               OutInt(X,4);OutInt(Y,4);OutImage;
            end;
         end;
      end;

      procedure Diagram(X1,Y1,X,Y,Going); name X1,Y1;
                                          integer X1,Y1,X,Y,Going;
      begin
         Boolean Ended;
         integer OldX, OldY, W;
         switch Sym_Action := H_Sym, S_Sym, E_Sym, D_Sym, Sy_Sym, L_Sym, R_Sym,
                              W_Sym, B_Sym, St_Sym, Su_Sym;

         if Diag(X1,Y1)<0 then Follow(X1,Y1,Going);

         while not Ended do
         begin
            GoTo Sym_Action(Diag(X1,Y1));

H_Sym:
            OutLine("Hold("&Form(X1,Y1,Period,2)&");",9);
            Next_Sym(X1,Y1);
            GoTo Done;

S_Sym:
            GoTo Done;

E_Sym:
            Ended := True;
            GoTo Done;

D_Sym:
```

```
                if Form(X1,Y1,Iden,2)<>NoText then
                     OutLine("! "&Form(X1,Y1,Iden,2)&";",9);
                OutLine("if "&Form(X1,Y1,Condition,2)&" then",9);
                OutLine("begin",9);
                OldX := X1; OldY := Y1+1;
                Diagram(OldX,OldY,X1,Y1,Downwards);
                if Diag(X1+1,Y1)<>0 or Diag(X1-1,Y1)<>0 then
                begin
                   OutLine("end else begin",9);
                   if Diag(X1+1,Y1)<>0 then
                   begin
                      X1 := X1+1;
                      W := Rightwards;
                   end else
                   begin
                      X1 := X1-1;
                      W := Leftwards;
                   end;
                   Diagram(X1,Y1,X1,Y1,W);
                end;
                OutLine("end;",9);
                Next_Sym(X1,Y1);
                GoTo Done;

Sy_Sym:
                begin
                   procedure Handle_Drop;
                   begin
                      if Form(X1,Y1,Iden,2)<>Notext then
                            OutLine("! "&Form(X1,Y1,Iden,2)&";",9);
                      if Diag(OldX,OldY)=Res_Sym then
                            OutLine(Form(OldX,OldY,Iden,2)&
                              ".Release("&Form(X1,Y1,Amount,2)&");",9)
                      else if Diag(OldX,OldY)=Bin_Sym then
                            OutLine(Form(OldX,OldY,Iden,2)&
                              ".Give("&Form(X1,Y1,Amount,2)&");",9)
                      else
                            OutLine(Form(OldX,OldY,Iden,2)&
                              ".Add("&Form(X1,Y1,Amount,2)&");",9);
                   end;

                   procedure Handle_Grab;
                   begin
                         if Form(X1,Y1,Iden,2)<>Notext then
                               OutLine("! "&Form(X1,Y1,Iden,2)&";",9);
                         if Diag(OldX,OldY)=Res_Sym then
                            OutLine(Form(OldX,OldY,Iden,2)&
                              ".Acquire("&Form(X1,Y1,Amount,2)&");",9)
                         else if Diag(OldX,OldY) = Bin_Sym then
                            OutLine(Form(OldX,OldY,Iden,2)&
                              ".Take("&Form(X1,Y1,Amount,2)&");",9)
                         else
                            OutLine(Form(OldX,OldY,Iden,2)&
                              ".Remove("&Form(X1,Y1,Amount,2)&");",9);
                   end;

                   if Diag(X1-1,Y1)<0 then
                   begin
                      OldX:=X1-1;OldY:=Y1;W:=Leftwards;
                      Follow(OldX,OldY,W);
                      if OldX<>X1-1 or OldY<>Y1 then Handle_Drop else
                      begin
                         Follow_Back(OldX,OldY,W);
                         if OldX<>X1-1 or OldY<>Y1 then Handle_Grab;
                      end;
                   end;
                   if Diag(X1+1,Y1)<0 then
                   begin
                      OldX:=X1+1;OldY:=Y1;W:=Rightwards;
                      Follow(OldX,OldY,Rightwards);
                      if OldX<>X1+1 or OldY<>Y1 then Handle_Drop else
                      begin
                         Follow_Back(OldX,OldY,Rightwards);
                         if OldX<>X1+1 or OldY<>Y1 then Handle_Grab
                      end;
                   end;
                   Y1:=Y1+1;
                   if Diag(X1,Y1)<0 then Next_Sym(X1,Y1);
                end;
                GoTo Done;

L_Sym:
                Ended := True;
                GoTo Done;

R_Sym:
                GoTo Done;

W_Sym:
                if Form(X1,Y1,Iden,2)<>NoText then
                     OutLine("! "&Form(X1,Y1,Iden,2)&";",9);
                OutLine("while "&Form(X1,Y1,Condition,2)&" do",9);
                OutLine("begin",9);
```

```
            Y1 := Y1+1;
            Diagram(X1,Y1,X1,Y1,Downwards);
            OutLine("end;",9);
            Next_Sym(X1,Y1);
            GoTo Done;

B_Sym:
            GoTo Done;

St_Sym:
            Go To Done;

Su_Sym:     Go To Done;

Done:
         end;
      end;

      inspect Sub_List do
      begin
         while not empty do inspect first do
         begin
            OutLine("%include "&Ident.Strip&".sim",6);
            Sub_List2.Into(Get);
         end;
      end;

      inspect Entity_List do
      begin
         while not empty do inspect first do
         begin
            integer Count;
            OutImage;
            OutLine("entity class "&Ident.Strip&"_C;",6);
            OutLine("begin",6);
            for Count := 0 step 1 until 2 do
            begin
               T :- Form(X,Y,Locals+Count,2).Strip;
               if T<>NoText then OutLine("   "&T,9);
            end;
            T :- Form(X,Y,Successor,2).Strip;
            if T<>NoText then
            begin
               Dist :- copy(Ident&"_A");
               OutLine("new "&Ident&"_C("""&Ident&
                       """).Schedule("&Dist&".Sample);",9);
               Dist_List.Into(new Dist_Item(X,Y,Dist,
                                    GetNext(T),GetNext(T),GetNext(T)));
            end;

            ! Process the activity diagram;

            ! Allow for several heads;
            X1 := X; Y1 := Y;
            while Diag(X1,Y1+1)=Start_Sym do Y1 := Y1+1;
            Y1 := Y1 + 1;

         ! This does the real work and is used recursively for nested branches;

            Diagram(X1,Y1,X,Y,Downwards);

            OutLine("end-of-"&First.Ident&";",6);
            OutImage;
             Entity_List2.Into(Get);
         end;
      end;
   end;

   procedure Print_Decls;
   begin
      inspect Entity_List2 do while not Empty do
      begin
         OutLine("ref("&First.Ident&"_C) "&First.Ident&";",6);
          Entity_List.Into(Get);
      end;
      inspect Sub_List2 do while not Empty do
      begin
         OutLine("ref("&First.Ident&"_C) "&First.Ident&";",6);
          Sub_List.Into(Get);
      end;
      inspect Res_List do while not Empty do
      begin
         OutLine("ref(Res) "&First.Ident&";",6);
         Res_List2.Into(Get);
      end;
      inspect Bin_List do while not Empty do
      begin
         OutLine("ref(Bin) "&First.Ident&";",6);
         Bin_List2.Into(Get);
      end;
      inspect Store_List do while not Empty do
      begin
         OutLine("ref(Store) "&First.Ident&";",6);
```

```
      Store_List2.Into(Get);
   end;
   inspect Dist_List do while not Empty do
   inspect First when Dist_Item do
   begin
      OutLine("ref("&Sort&") "&Ident&";",6);
      Dist_List2.Into(Get);
   end;
end;

procedure Print_News;
begin
   ! Print out the object generation statements;
   inspect Entity_List do while not Empty do
   begin
      OutLine(First.Ident&" :- new "&First.Ident&"_c("""
            &First.Ident&""");",6);
      Entity_List2.Into(Get);
   end;
   inspect Res_List2 do while not Empty do
   begin
      inspect First do
         OutLine(Ident&" :- new Res("""
               &Ident&""", "&Form(X,Y,Amount,2)&");",6);
      Res_List.Into(Get);
   end;
   inspect Bin_List2 do while not Empty do
   begin
      inspect First do
         OutLine(Ident&" :- new Bin("""
               &Ident&""", "&Form(X,Y,Amount,2)&");",6);
      Bin_List.Into(Get);
   end;
   inspect Store_List2 do while not Empty do
   begin
      inspect First do
         OutLine(Ident&" :- new Store("""
               &Ident&""", "&Form(X,Y,Amount,2)&");",6);
      Store_List.Into(Get);
   end;
   inspect Dist_List2 do while not Empty do
   begin
      inspect First when Dist_Item do
      begin
         OutText("      "&Ident&" :- new "&Sort&"("""
               &Ident&""", "&P1);
         if P2<>NoText then OutText(","&P2);
         OutLine(");",0);
      end;
      Dist_List.Into(Get);
   end;
end;

procedure Print_Schedules;
begin
   ! Schedule the initial entities;
   inspect Entity_List2 do while not Empty do
   begin
      inspect First do
         OutLine(Ident&".Schedule("&Form(X,Y,Sched,2)&");",6);
      Entity_List.Into(Get);
   end;
   inspect Sub_List2 do while not Empty do
   begin
      inspect First do
         OutLine(Ident&".Schedule("&Form(X,Y,Sched,2)&");",6);
      Sub_List.Into(Get);
   end;
end;

Open(Blanks(80));
Entity_List :- new Global_List;
Entity_List2 :- new Global_List;
Sub_List :- new Global_List;
Sub_List2 :- new Global_List;
Res_List :- new Global_List;
Res_List2 :- new Global_List;
   Bin_List :- new Global_List;
   Bin_List2 :- new Global_List;
   Store_List :- new Global_List;
   Store_List2 :- new Global_List;
   Dist_List :- new Global_List;
   Dist_List2 :- new Global_List;

   Prologue;
   Find_Globals;
   Build_Entities;
   Print_Decls;
   Print_News;
   Print_Schedules;
   Epilogue;
   Close;
  end+inspect+model;
```

```
  end-procedure-Gen;


  procedure Read_Diag(F); ref(InFile) F;
  begin
      integer X, Y, I;
      inspect F do
      while not LastItem do
      begin
         X := InInt; Y := InInt;
         Diag(X,Y) := InInt;
          InImage;
          for I := 1 step 1 until 6 do
          begin
             Form(X,Y,I,2) :- Copy(Image.Strip);
             InImage;
          end;
      end;
   end;

  procedure Draw_Diag;
  begin
     integer X,Y;
     for X := 0 step 1 until XSquares-1 do
        for Y := 0 step 1 until 11 do
        begin
           Display_Square(Diag(X,Y),Y*20,X*20,Y*20,X*20);
           SetColour(White);
           DrawSquare(Y*20+15,X*20,20,20);
        end;
  end;

! Define the basic symbols for the diagrams;

  procedure Print_Sym(Symbol,Y,X,Colour); integer Symbol, Y, X, Colour;
  begin
     switch PSYM := HoldSym, StartSym, EndSym, DecisionSym, SynchSym, LinkSym,
                    ResSym, WhileSym, BinSym, StoreSym, SubSym,
                      LRLink, RLLink, DULink, UDLink, RULink, RDLink,
                    LULink, LDLink, URLink, DRLink, ULLink, DLLink;
     integer Old_Colour;
     if Symbol<> 0 and then Symbol>= DL  and then Symbol<=Max_Sym then
     begin
        if Symbol<0 then Symbol := Max_Sym - Symbol;
        Old_Colour := SetColour(Colour);
        GoTo PSYM(Symbol);

HoldSym:    DrawSquare(Y+17,X+2,16,16);
            goto Done;

StartSym:   DrawLine(Y+21,X+3,0,14);
            DrawSector(Y+21,X+10,7,500,500,0);
            goto Done;

EndSym:     DrawLine(Y+29,X+5,0,14);
            DrawSector(Y+29,X+10,7,0,500,0);
            goto Done;

DecisionSym: DrawLine(Y+18,X+10,7,-8);
            DrawLine(Y+18,X+10,7,8);
            DrawLine(Y+25,X+2,7,8);
            DrawLine(Y+25,X+18,7,-8);
            goto Done;

SynchSym:   DrawLine(Y+25,X+3,0,14);
            DrawLine(Y+15,X+10,20,0);
            DrawCircle(Y+25,X+10,3);
            goto Done;

LinkSym:    DrawLine(Y+25,X+3,0,14);
            DrawLine(Y+15,X+10,20,0);
            goto Done;

ResSym:     DrawCircle(Y+25,X+10,7);
            goto Done;

WhileSym:   DrawSector(Y+25,X+10,7,750,750,0);
            SetColour(if X>480 then Green else LGreen);
            DrawLine(Y+25,X+3,0,7);
            DrawLine(Y+25,X+10,7,0);
            SetColour(Colour);
            DrawLine(Y+32,X+10,3,3);
            DrawLine(Y+32,X+10,-3,3);
            goto Done;

BinSym:     DrawLine(Y+18,X+3,15,3);
            DrawLine(Y+18,X+17,15,-3);
            DrawLine(Y+33,X+6,0,8);
            goto Done;

StoreSym:   DrawSquare(Y+19,X+4,16,12);
            DrawLine(Y+23,X+4,0,12);
            DrawLine(Y+27,X+4,0,12);
```

```
            DrawLine(Y+32,X+4,0,12);
            goto Done;

SubSym:     DrawSquare(Y+23,X+6,8,8);
            goto Done;

LRLink:     DrawLine(Y+25,X,0,20);
            DrawLine(Y+21,X+16,4,4);
            GoTo Done;

RLLink:     DrawLine(Y+25,X,0,20);
            DrawLine(Y+29,X+4,-4,-4);
            goto Done;

DULink:     DrawLine(Y+15,X+10,20,0);
            DrawLine(Y+18,X+13,-3,-3);
            goto Done;

UDLink:     DrawLine(Y+15,X+10,20,0);
            DrawLine(Y+32,X+7,3,3);
            goto Done;

RULink:     DrawSector(Y+15,X,10,750,250,0);
            goto Done2;

RDLink:     DrawSector(Y+35,X,10,0,250,0);
            goto Done2;

LULink:     DrawSector(Y+15,X+20,10,500,250,0);
            goto Done2;

LDLink:     DrawSector(Y+35,X+20,10,250,250,0);
            goto Done2;

URLink:     DrawSector(Y+35,X+20,10,250,250,0);
            goto Done2;

DRLink:     DrawSector(Y+15,X+20,10,500,250,0);
            goto Done2;

ULLink:     DrawSector(Y+35,X,10,0,250,0);
            goto Done2;

DLLink:     DrawSector(Y+15,X,10,750,250,0);
            goto Done2;

Done2:      SetColour(White);
            DrawSquare(Y+15,X,20,20);

Done:       SetColour(Old_Colour);
        end;
    end-Print_Sym;

    procedure Set_Forms;
    begin

        Titles(Hold_Sym,1) :- "Reason: ";
        Titles(Hold_Sym,2) :- "Delay: ";

        Titles(Start_Sym,1) :- "Name: ";
        Titles(Start_Sym,2) :- "Scheduling: ";
        Titles(Start_Sym,3) :- "Successor: ";
        Titles(Start_Sym,4) :- "Locals: ";
        Titles(Start_Sym,5) :- "Locals: ";
        Titles(Start_Sym,6) :- "Locals: ";

        Titles(Res_Sym,1) :- "Name: ";
        Titles(Res_Sym,2) :- "Amount: ";

        Titles(Decision_Sym,1) :- "Reason: ";
        Titles(Decision_Sym,2) :- "Condition: ";

        Titles(Synch_Sym,1) :- "Notes: ";
        Titles(Synch_Sym,2) :- "Amount: ";

        Titles(While_Sym,1) :- "Reason: ";
        Titles(While_Sym,2) :- "Condition: ";

        Titles(Bin_Sym, 1) :- ("Name: ");
        Titles(Bin_Sym, 2) :- ("Amount: ");

        Titles(Store_Sym,1) :- "Name: ";
        Titles(Store_Sym,2) :- "Amount: ";

        Titles(Sub_Sym,1) :- "Name: ";
        Titles(Sub_Sym,2) :- "Source file: ";
        Titles(Start_Sym,3) :- "Parameters: ";

    end;

    procedure Draw_Button(Y,Colour,Action);
                    integer Y,Colour; text Action;
    begin
```

```
    SetColour(Colour);
    FillSquare(Y,575,55,55);
    SetPos(TextLine(Y+35),TextPos(580));
    OutText(Action);
end;

procedure Display_Square(Symbol,NewY,NewX,OldY,OldX);
                integer Symbol, NewY,NewX,OldY,OldX;
begin
    integer D, S;

    if OldX<500 then
    begin        ! Started on the canvas;
       SetColour(LGreen);
       if not No_Move then
       begin         ! Restore the square moved from;
          S := Diag(OldX//20,OldY//20);
          FillSquare(OldY+15,OldX,20,20);
          if S<>0 then
          begin
             if S>0 then Print_Sym(S,OldY,OldX,Blue) else
             begin
                D := -1; S := -S;
                while S<>0 do
                begin
                   if ((S//2)*2)<>S then Print_Sym(D,OldY,OldX,Blue);
                   S := S//2;
                   D := D - 1;
                end;
             end;
          end;
       end else FillSquare(NewY+15,NewX,20,20);
       SetColour(White);
       DrawSquare(OldY+15,OldX,20,20);
       if NewX<500 then
       begin
          if Symbol>0 then Print_Sym(Symbol,NewY,NewX,Blue) else
          begin
             D := -1; Symbol := -Symbol;
             while Symbol<>0 do
             begin
                if ((Symbol//2)*2)<>Symbol then Print_Sym(D,NewY,NewX,Blue);
                Symbol := Symbol//2;
                D := D - 1;
             end;
          end;
       end;
    end else
    if (OldX>=500 and then OldX<560) then
    begin         ! Started on the menu area;
       if OldY>=40 then
        begin
           SetColour(White);
          DrawSquare(OldY+15,OldX,20,20);
        end;
        if NewX<560 then
       begin         ! Moved to the canvas or menu area;
          SetColour(Red);
          DrawSquare(NewY+15,NewX,20,20);
       end else
       begin       ! Movedto the button area;
          if NewY<120 then Draw_Button(55,Red,"Exit")
                      else Draw_Button(120,Red,"Generate");
       end
    end else
    begin          ! Started on the button area;
       if OldY<120 then Draw_Button(55,Green,"Exit")
                      else Draw_Button(120,Green,"Generate");
        if NewX<560 then
       begin         ! Moved to the menu area;
          SetColour(Red);
          DrawSquare(NewY+15,NewX,20,20);
       end else
       begin       ! Movedto the button area;
          if NewY<120 then Draw_Button(55,Red,"Exit")
                      else Draw_Button(120,Red,"Generate");
       end
    end;
end;

procedure Draw_Link(OldY,OldX,NewY,NewX,PrevSymb);
    name PrevSymb; integer OldY,OldX,NewY,NewX,PrevSymb;
begin
    ! Draw a link from one square to another. Must be adjacent;
    integer NewDirection;
    NewDirection := OldY-NewY+2*(OldX-NewX);
    PrevSymb := Lnk(NewDirection//20,PrevSymb);
    if First_Link then
    begin
       First_Link := False;
       Display_Square(Diag(OldX//20,OldY//20),OldY,OldX,OldY,OldX);
       SetColour(White);
       DrawSquare(OldY+15,OldX,20,20);
```

```
      end;
      Diag(OldX//20,OldY//20) := Diag(OldX//20,OldY//20) + SymbMap(PrevSymb);
      Print_Sym(PrevSymb,OldY,OldX,Blue);
   end;

! Initialise the links table;

   Lnk(Up,0) :=DU; Lnk(Dn,0) :=UD; Lnk(Lf,0) :=RL; Lnk(Rt,0) :=LR; Lnk(NW,0) :=0;
   Lnk(Up,LR):=RU; Lnk(Dn,LR):=RD; Lnk(Lf,LR):=Del;Lnk(Rt,LR):=LR; Lnk(NW,LR):=0;
   Lnk(Up,RL):=LU; Lnk(Dn,RL):=LD; Lnk(Lf,RL):=RL; Lnk(Rt,RL):=Del;Lnk(NW,RL):=0;
   Lnk(Up,DU):=DU; Lnk(Dn,DU):=Del;Lnk(Lf,DU):=UL; Lnk(Rt,DU):=UR; Lnk(NW,DU):=0;
   Lnk(Up,UD):=Del;Lnk(Dn,UD):=UD; Lnk(Lf,UD):=DL; Lnk(Rt,UD):=DR; Lnk(NW,UD):=0;
   Lnk(Up,RU):=DU; Lnk(Dn,RU):=Del;Lnk(Lf,RU):=UL; Lnk(Rt,RU):=UR; Lnk(NW,RU):=0;
   Lnk(Up,RD):=Del;Lnk(Dn,RD):=UD; Lnk(Lf,RD):=DL; Lnk(Rt,RD):=DR; Lnk(NW,RD):=0;
   Lnk(Up,LU):=DU; Lnk(Dn,LU):=Del;Lnk(Lf,LU):=UL; Lnk(Rt,LU):=UR; Lnk(NW,LU):=0;
   Lnk(Up,LD):=Del;Lnk(Dn,LD):=UD; Lnk(Lf,LD):=DL; Lnk(Rt,LD):=DR; Lnk(NW,LD):=0;
   Lnk(Up,UR):=RU; Lnk(Dn,UR):=RD; Lnk(Lf,UR):=Del;Lnk(Rt,UR):=LR; Lnk(NW,UR):=0;
   Lnk(Up,DR):=RU; Lnk(Dn,DR):=RD; Lnk(Lf,DR):=Del;Lnk(Rt,DR):=LR; Lnk(NW,DR):=0;
   Lnk(Up,UL):=LU; Lnk(Dn,UL):=LD; Lnk(Lf,UL):=RL; Lnk(Rt,UL):=Del;Lnk(NW,UL):=0;
   Lnk(Up,DL):=LU; Lnk(Dn,DL):=LD; Lnk(Lf,DL):=RL; Lnk(Rt,DL):=Del;Lnk(NW,DL):=0;

! Create a map from symbols to stored symbols;

   SymbMap(LR):=L_R; SymbMap(RL):=R_L; SymbMap(DU):=D_U; SymbMap(UD):=U_D;
   SymbMap(RU):=R_U; SymbMap(RD):=R_D; SymbMap(LU):=L_U; SymbMap(LD):=L_D;
   SymbMap(UR):=U_R; SymbMap(DR):=D_R; SymbMap(UL):=U_L; SymbMap(DL):=D_L;

! Create the main canvas window;

   Set_Forms;
   SetColour(LGreen);
   FillSquare(0,0,255,500); SetPos(1,TextPos(100));
   OutText("Demos graphical input");

! Create the menu window;

      SetColour(Green);
      FillSquare(0,500,255,60);
      SetPos(1,TextPos(515));
      OutText("Menu");

! Add the symbols to the menu window;

  for I := Hold_Sym step 1 until Store_Sym do
      Print_Sym(I,I*20+20,500,Yellow);

! Add the links to the menu window;

      Print_Sym(LR,20+20,520,Yellow);

! Add sub-models to the menu window;

      Print_Sym(Sub_Sym,40+20,520,Yellow);

   SetColour(White);
   for I := 55 step 20 until 255 do Drawline(I,500,0,60);
   DrawLine(0,500,255,0);
   DrawLine(55,520,200,0);
   DrawLine(55,540,200,0);
   DrawLine(0,560,255,0);

! Set up the control panel;

   SetPos(1,TextPos(570));
   OutText("Controls");

   Draw_Button(55,Green,"Exit");       ! Make the Exit button;
   Draw_Button(120,Green,"Generate");  ! Make the Generate button;

! Draw the grid;

   SetColour(White);
   for I := 0 step 20 until 500 do DrawLine(15,I,240,0);
   for I := 15 step 20 until 255 do DrawLine(I, 0, 0, 500);

! Check for a file to load;

   SetPos(TextPos(280),1);
   OutText("Give input file name (Type 'Cr' for no)");
   F_Name :- InText(40);
   if F_Name <> NoText then
   begin
      InF :- new InFile(F_Name);
      InF.Open(Blanks(80));
      Read_Diag(InF);
      No_Move := True;
      Draw_Diag;
      No_Move := False;
   end;

   X := 500;  Y := 40;
   SetColour(Red);
   DrawSquare(Y+15,X,20,20);
```

```
  CurrSymb := Hold_Sym;
  Linking := False;

! The main input loop;

  while not Exited do
  begin

! First check for cursor key presses;
    Current_Char := InChar;
    if Current_Char='1' then Exited:=True else
    if Current_Char=Char(MoveR) then ! Move right;
    begin
      X := if X<540 then X + 20 else 580;
      if Linking then Draw_Link(Y,X-20,Y,X,PrevSymb)
                 else Display_Square(CurrSymb,Y,X,Y,if X=580 then 540 else X-20);
      No_Move := False;
    end else
    if Current_Char=Char(MoveL) then ! Move left;
    begin
      X := if X>540 then 540 else if X>20 then X-20 else 0;
      if Linking then Draw_Link(Y,X+20,Y,X,PrevSymb)
                 else Display_Square(CurrSymb,Y,X,Y,X+20);
      No_Move := False;
    end else
    if Current_Char=Char(MoveD) then ! Move down;
    begin
      Y := if Y<200 then Y+20 else 220;
      if Linking then Draw_Link(Y-20,X,Y,X,PrevSymb)
                 else Display_Square(CurrSymb,Y,X,Y-20,X);
      No_Move := False;
    end else
    if Current_Char=Char(MoveU) then ! Move up;
    begin
      if X>=500 then Y := if Y>60 then Y-20 else 40
                else Y := if Y>20 then Y-20 else 0;
      if Linking then Draw_Link(Y+20,X,Y,X,PrevSymb)
                 else Display_Square(CurrSymb,Y,X,Y+20,X);
      No_Move := False;
    end else
    if Current_Char=Char(Left) then ! Change the current symbol or insert it;
    begin
      if X>=500 and then X<560 and then Y<240 then
      begin
         if X<520 then
         begin
             CurrSymb := Y//20 - 1
         end else if Y>40 then CurrSymb := Sub_Sym else
         begin
            CurrSymb := -Y//20 + 1;
            Linking := False;
         end
      end else

! Add or delete a symbol;

      if Y<240 and then X<500 then !  Set the symbol at the current position;
      begin
          if CurrSymb<0 then
          begin
             Linking := not Linking;
             First_Link := Linking;
             PrevSymb := 0;
          end else
           begin
             SetColour(Black);
             Diag(X//20, Y//20) := CurrSymb;
              Display_Square(CurrSymb,Y,X,Y,X);
             No_Move := True;
         end
      end else

! Control panel button pressed;
      if X>=560 then   ! Quit;
      begin
         if Y<120 then Exited := True else Gen;
      end
    end else
    if Current_Char=Char(Middle) then   ! Delete current symbol;
    begin
      if X<500 and then Y<240 then
      begin
          Diag(X//20, Y//20) := 0;
           SetColour(LGreen);
          FillSquare(Y+15,X,20,20);
           SetColour(Red);
           DrawSquare(Y+15,X,20,20);
           Print_Sym(CurrSymb,Y,X,Blue);
      end
    end else
    if Current_Char=Char(Right) then   ! Enter form attributes;
    begin
      SetColour(LGreen);
```

```
      FillSquare(260,0,85,500);
      J := TextLine(280);
      if Diag(X//20,Y//20)<>0 then for I := 0 step 1 until 5 do
      begin
         if Titles(Diag(X//20,Y//20),I+1)=/= notext then
         begin
             SetPos(J+I,1);
            Form(X//20,Y//20,I+1,1) :- Titles(Diag(X//20,Y//20),I+1);
            OutText(Titles(Diag(X//20,Y//20),I+1));
            SetPos(J+I,20);
            OutText(Form(X//20,Y//20,I+1,2)&" ");
            String :- InText(40).Strip;
            if String<>NoText then Form(X//20,Y//20,I+1,2) :- String;
         end;
      end;
   end;
      ox := x; oy := y; ob := Button; Prev_Char := Current_Char;
  end;


end;

! Write out the matrix;
  OutText("Which file for saving the model?"); BreakOutImage;
  InImage;
  F_Name :- copy(SysIn.Image.Strip);
  if F_Name<>NoText then OF :- new OutFile(F_Name);
  inspect OF do
  begin
     Open(blanks(80));
     for I := 0 step 1 until YSquares do
     begin
        for J := 0 step 1 until XSquares do if Diag(j,i) ne 0 then
        begin
           outint(j,8);outint(i,8);
           outint(Diag(j,i),14); outimage;
           for d := 1 step 1 until 6 do
           begin
               outtext(Form(j,i,d,2));
               outimage
           end;    ! of one form;
           outimage;
        end;    ! of one symbol;
        outimage;
     end;    ! of the grid;
     Close;
  end;
  if Inf=/=None then InF.Close;
end
```

Appendix B

Demos Models and Traces

This Appendix contains the DEMOS source and, where appropriate, output of some of the models used in Chapters 3, 4, 5 and 6 of this dissertation.

Chapter 3

Figure 3.4

```
Entity class Seq;
begin
   Hammer.Acquire(1);
   Hold(3);
   Hammer.Release(1);
end;
```

Figure 3.5

```
Entity class Seq;
begin
   while True do
   begin
      Hammer.Acquire(1);
      Hold(3);
      Hammer.Release(1);
   end;
end;
```

Figure 3.6

```
Entity class Seq;
begin
   integer Val;
   Val := 4;
   while True do
   begin
      Val := Val + 2;
      Hold(3);
      Val := Val * 2;
   end;
end;
```

Figure 3.7

```
Entity class Seq;
begin
   integer Val;
   Val := 4;
   while True do
   begin
      Val := Val + 2;
      Hold(3);
      if Val<10 then Val := Val * 2 else Val := 4;
   end;
end;
```

Figure 3.8

```
Entity class Seq;
begin
   integer Val;
   Val := 4;
   while Val<10 do
   begin
      Val := Val + 2;
      Hold(3);
   end;
end;
```

Figure 3.9

```
Entity class Station;
begin
   while True do
   begin
      new Packet.Schedule(3.0);
      Hold(2.0);
   end;
end;

Entity class Packet;
begin
end;
```

Figure 3.10

```
Entity class Station;
begin
   while True do
   begin
      P1.Schedule(3.0);
      Hold(2.0);
   end;
end;

Entity class Packet;
begin
   Passivate;
end;

ref (Packet) P1;
P1:- new Packet("P1");
```

Figure 3.11

```
entity class Ship_C;
begin
   new Ship.Schedule(4);
   ! grab 2 tugs;
   Tugs.Acquire(2);
   ! and a jetty;
   Jetties.Acquire(1);
   Hold(3);
   ! let the tugs go;
   Tugs.Release(2);
   Hold(10);
   ! ready to leave;
   Tugs.Acquire(1);
   Hold(3);
   ! clear of jetty;
   Jetties.Release(1);
   ! gone away;
   Tugs.Release(1);
end-of-Ship;

ref(Res) Jetties, Tugs;

Ship :- new Ship_c("Ship");
Tugs :- new Res("Tugs", 3);
Jetties :- new Res("Jetties", 2);
```

Figure 3.13

```
Entity class Producer;
begin
   while True do
   begin
      Hold(Make_Time);
      Wid.Give(1);
   end;
end;

Entity class Consumer;
begin
   while True do
   begin
      Wid.Take(1);
      Hold(Finish_Time);
   end;
end;

ref(Bin) Wid;

Wid :- new Bin("Widgets",0);
```

Figure 3.15

```
Entity class Producer;
begin
   Hold(Make_Time);
   Widgets.Add(1);
   repeat;
end;

Entity class Consumer;
begin
   Widgets.Remove(1);
   Hold(Finish_Time);
   repeat;
end;

ref(Store) Widgets;

Widgets :- new Store("Widgets",4,0);
```

Figure 3.17

```
Entity class Car;
begin
   new Car("Car").Schedule(ArrivalTime);
   Hold(TripTime1);
   FerryQueue.Wait;
end;

Entity class Ferry;
begin
   ref(Car) Cargo;
   while True do
   begin
      Cargo :- FerryQueue.Coopt;
      Hold(VoyageTime1);
      Cargo.Schedule(0);
      Hold(VoyageTime2);
   end;
end;

ref(WaitQ) FerryQueue;

FerryQueue:- new WaitQ("Ferries");
```

Figure 3.19

```
Entity class Waiter;
begin
   CQ.WaitUntil(Val>3);
end;

Entity class Signaller;
begin
   while True do
   begin
      Val := Val + 1;
      CQ.Signal;
   end;
end;

integer Val;

ref(CondQ) CQ;
CQ :- new CondQ("CQ");
Figure 3.22
Entity class Interrupted;
begin
   Hold(TDo);
   if Interrupt=3 then new
Interrupted("Ited").Schedule(0);
end;

Entity class Interrupter;
begin
   Ited.Interrupt(3);
end;

Ited :- new Interrupted("Ited");
Iter :- new Interrupter("Iter");
```

Figure 3.24

```
EXTERNAL class DEMOS;
DEMOS class E_DEMOS;
begin

   Entity class Philosopher(Right_Fork,Left_Fork,T_Feed, T_Think);
              ref(Res) Right_Fork,Left_Fork; REAL T_Feed,T_Think;
   begin
      while True do
      begin
         Right_Fork.acquire(1);
         Hold(0.2);
         Left_Fork.acquire(1);
         Hold(T_Feed);
         Right_Fork.release(1);
         Left_Fork.release(1);
         Hold(T_Think);
      end;
   end of Philosopher;

end of E_DEMOS;

begin
   EXTERNAL class E_DEMOS;
   E_DEMOS
   begin
      ref(Res) Fork1, Fork2, Fork3;
      real I_T_Feed, I_T_Think;

      I_T_Feed := InReal; I_T_Think := InReal;

      Fork1 :- new Res("Fork",1);
      Fork2 :- new Res("Fork",1);
      Fork3 :- new Res("Fork",1);
  new Philosopher("P",Fork1,Fork2,I_T_Feed,I_T_Think).Schedule(0);
  new Philosopher("P",Fork2,Fork3,I_T_Feed,I_T_Think).Schedule(0);
  new Philosopher("P",Fork3,Fork1,I_T_Feed,I_T_Think).Schedule(0);
      Hold(100.0);
   end;
end
```

Figure 3.26

```
begin
   external class demos;
   DEMOS
   begin

      entity class Ship_C;
      begin
         new Ship.Schedule(4);
         ! grab 2 tugs;
         Tugs.Acquire(2);
         ! and a jetty;
         Jetties.Acquire(1);
         Hold(3);
         ! let the tugs go;
         Tugs.Release(2);
         Hold(10);
         ! ready to leave;
         Tugs.Acquire(1);
         Hold(3);
         ! clear of jetty;
         Jetties.Release(1);
         ! gone away;
         Tugs.Release(1);
      end-of-Ship;

      ref(Ship_C) Ship;
      ref(Res) Jetties;
      ref(Res) Tugs;
      Ship :- new Ship_c("Ship");
      Tugs :- new Res("Tugs", 3);
      Jetties :- new Res("Jetties", 2);
      Ship.Schedule(0.0);

      Hold(100);
   end
end
```

Chapter 4

Chapter 5

Figure 4.1

```
begin
   external class demos;
   DEMOS
   begin

      entity class Ship_C;
      begin
         new Ship.Schedule(4);
         ! grab a jetty;
         Jetties.Acquire(1);
         ! grab 2 tugs;
         Tugs.Acquire(2);
         Hold(3);
         ! let the tugs go;
         Tugs.Release(2);
         Hold(10);
         ! ready to leave;
         Tugs.Acquire(1);
         Hold(3);
         ! clear of jetty;
         Jetties.Release(1);
         ! gone away;
         Tugs.Release(1);
      end-of-Ship;

      ref(Ship_C) Ship;
      ref(Res) Jetties;
      ref(Res) Tugs;
      Ship :- new Ship_c("Ship");
      Tugs :- new Res("Tugs", 3);
      Jetties :- new Res("Jetties", 2);
      Ship.Schedule(0.0);

      Hold(100);
   end
end
```

Figure 4.8

```
begin
   external class DEMOS;
   DEMOS
   begin

      Entity class Host_c(PQ); ref(WaitQ) PQ;
      begin
         ref(File_c) F1;
         new File_c.Into(PQ);
         while True do
         begin
            F1 :- PQ.coopt;
            while Printer.Avail=0 do Hold(0.01);
            Printer.Acquire(1);
            Hold(4.0);
            Printer.Release(1);
            Hold(1.0);
         end;
      end;

      Entity class File_c(PQ); ref(WaitQ) PQ;
      begin
         new File_c("File").Schedule(2.0);
         PQ.Wait;
      end;

      ref(Res) Printer;

      Printer :- new Res("Printer",1);

      for I := 1 step 1 until InInt do
         new Host_c("Host",new WaitQ).Schedule(0.0);
      Hold(InReal);
   end;
end
```

Figure 4.9

```
external class DEMOS;
DEMOS
begin
   character Ch;
   Boolean Refresh,
           TraceOn,
           ReportOn,
           Contention;
   integer I,
           Small, Medlum,
           Threhold,
           NumberofXmit;
   ref(Count) NumberofAttempt,
              NumofContention,
              NumberofFailures,
              NumberofSuccess;
   long real SimTime
             Arrivaload,
             RefreshTime,
             BackOffScale;

    ref(WaitQ) EtherQ;!For transmitter waiting for ether to clear;
    ref(CondQ) Packet6;
    ref(Channel) Ether;
! Reporting and tracing;

    procedure TraceImage(T,N); text T; real N;
    begin
    if TraceOn then
       begin
          OutText(T);
          OutFix(N, 2, 12);
          OutImage;
       end
       end..of..TraceImage;

    Grocedure ReportEvent(Mess, Num); text Mess; integer Num;
     begin
     if TraceOn then
          OutText("Time  ");OutFix(Time,2,10); OutText("  ");
          OutText(Mess); if Num>0 then  OutText(", b XMitter");
          OutInt(Abs(Num),12);
          OutImage;
          end..of..ReportEvent;
!
 Ethernet itself - state variables etc. ;

    entity class Channel;
    begin
    long real LastTime;
    Boolean Busy;
Loop:
         Cancel;
         NumberofXmits := 0;
```

```
    ! Allow them to try;
    while EtherQ.Length>0 do EtherQ.Coopt.Schedule(0.0);
        Hold(0.0);                          ! Go to back of event list;
         Contention := NumberofXmits>1;
         if Contention then
         begin
            reportEvent("Contention level ",-NumberofXMits);
            NumofContentions.Update(1);
         end;
      end..of..Channel;

      entity class Transmitter(InQ,N); ref(Queue) InQ; integer N;
      begin
      ref(Packet) Pkt;
      ref(IDist) Del1  Del2;
      integer NTries,Mask;
      Del1 :- new RandInt(Edit("Delay",N),1,255);
Loop:
         if InQ.Length=0 then PacketQ.WaitUntil(InQ.Length>0);
Loop2:
         if Ether.Busy then EtherQ.Wait;
         NumberofAttempts.Update(1);
         NumberofXmits :- NumberofXmits + 1;    ! Attempts at this
time;
         Ether.Busy := True;
         Hold(1.8);
         if Contention then
         begin
              if NTries<16 then
            begin
            inspect Ether do
            begin
               Busy := False
               if Idle then Schedule(0.0);
             end;
               Mask := Mask*2 + 1     ! Right shifted, one filled;
               Hold(mod(Del1.Sample,Mask+1));
               NTries := NTries + 1;
               goto Loop2
            end else begin
               ! Abandon;
               NTries  := 1;
               Mask := 0;
               inspect InQ.First when Packet do
               begin
                   NumberofFailures.Update(1):
                 ReportEvent("Packet abandoned");
                 Failed := True;
                 Schedule(0.0)
                 end;
                 inspect Ether do
                 begin
                 Busy := False;
                 if Idle then Schedule(0.0);
                 end;
             end;
        end else begin
```

```
          ! Transmit
          inspect InQ.First when Packet do if Size>1 then
Hold(Size-i);
          NTries := 1;
          Mask := 0;
          inspect InQ.First when Packet do
          begin
              Out;
                Failed := False;
             Schedule(0.0);
             NumberofSuccesses.Update(1);
             ReportEvent("Packet transmitted",N);
          end;
          inspect Ether do
          begin
             Busy := False;
             if idle then Schedule(0.0)
          end;
          Hold(0.0);
        end;
        repeat;
      end..of..Transmitter;

! Packet generation, one per transmitter;

      entity class Source(N); integer N;
      begin
      integer Size;
      real Choice;
      ref(RDist) Uni, Sizes1,Sizes2,Sizes3;
         ref(Queue) MyQ;
         ref(Packet) Pkt;
         ref(RDist) MyDelay;
         MyQ :- new Queue(Edit("Input",N));
         MyDelay :- new NegExp(Edit("ArrTime",N),i/ArrivalLoad);
         Uni :- new Uniform("Uni",0 180)
         Sizes1 :- new NegExp("Siz1",1/168);
         Sizes2 :- new NegExp("Siz2",1/1000);
         Sizes3 :- new NegExp("Siz3",1/80000);
         new Tranmitter(Edit("XMitter",N),MyQ,N).schedule(0);

         Hold(MyDelay.Sample);
         Choice := Uni.Sample;
         Size := if Choice<Small then Size1.Sample
            else if Choice<Medium then Sizes2.Sample
            else Sizes3.Sample;
         Pkt :- new  Packet(Edit("Packet",N),MyQ.Size);
         ReportEvent("Packet for transmision",A);
         TraceImage("Packet size is ",Size);
         Pkt.Schedule(0.0);
      end..of..source;

      entity class Packet(Q,Size); ref(Queue) Q; integer Size;
      begin
            real ArrTime;
        Boolean Failed;
        ArrTime := Time;
        PacketQ.Signal;
      end..of..Packet;
```

```
    OutText("Howlong for this run?"); OutImage;
    SimTime := InReal;
    OutText("Tracing? /n"); OutImage;
    InImage;
    Ch := InChar;
    if Ch-'T' then Trace else TraceOn := Ch='y' or Ch='Y';
    OutText("Percentages for Small and Medium?");OutImage;
    Small := InInt;  Medium := InInt;
    OutText("Threshold for initiating transmission?");outimage;
    Threshold := InInt;
    OutText("Arrival rate of packets?");OutImage;
    ArrivalLoad := InReal;
    OutText("Data arrives at rate - ")
    OutReal((Small*180+(Medium-Small*1000+
                      (100-Medium)*80000)/ArrivalLoad/100, 4,12);
    OutImage;
    OutText("Refresh time?");
    OutImage;
    RefreshTime := InReal
    OutText("Back off scale?"); OutImage;
    BackOffScale := InReal;

    OutF :- new PrintFile("ether.tra");
    OutF.Open(Blanks(80));

    PacketQ :- new CondQ("PacketQ");
    PacketQ.All :=True;
    NumberofAttempts :- new Count("Attempts");
    NumofContentions :- new Count("Contentions");
    NumberofFailures :- new Count("Failures")
    NumberofSuccesses :- new Count("Successes");
    EtherQ :- new WaitQ("EtherQ")
    for I := 1 step 1 until 10 do
      new Source(Edit("Source",I),I).Schedule(0.0);
    Ether Schedule(0.0);
    if Time<SimTime then
       Hold(if SimTime<RefreshTime then SimTime else RefreshTime);
    while Time<SimTime do
    begin
      QueueQ.Report;
      CondQQ.Report;
      CountQ.Report;
      Outf qua printfile .eject(1);
      Hold(RefreshTime);
    end;
  end--of--DEMOS--block
  end++of++program
```

Figure 4.10

```
BEGIN EXTERNAL CLASS DEMOS;
DEMOS
  Begin
    Ref(Res)Buffers;

    Entity Class Reader;
    Begin
      ! Read;
      Buffers.Acquire(1);
      Hold(2.0);
      Buffers.Release(1);
    ! Use;
      Hold(5.0);
      Repeat;
    End***Reader***;

    Entity Class Writer;
    Begin
    ! Gather;
      Hold(5.0);
    ! Write;
      Buffers.Acquire(3);
      Hold(3.0);
      Buffers.Release(3);
      Repeat;
    End***Writer***;

    Trace;
    Buffers   :- New Res("Buffers", 3);
    New Reader("R").Schedule(0.0);
    New Reader("R").Schedule(2.0);
    New Writer("W").Schedule(1.0);
    Hold(25.0);
  End;
End;
```

Chapter 5

The implmentation of M_SIM

```
simset class msim;
begin
   text procedure Join(T,F); text T; real F;
   begin
      text FT;
      FT :- Blanks(8);
      FT.PutFix(F,2);
      Join :- T&" "&FT;
   end;

   procedure Dump_Event_List;
   begin
      ref(Proc) P;
      P :- Event_List.First;
      while P=/= none do
      begin
         OutText(P.Title);
         OutFix(P.Ev_Time,2,8);
         OutImage;
         P :- P.Suc;
      end;
   end;

   Boolean Trace_Flag,
           Dump_Flag;
   ref(Head) Event_List;
   ref(Proc) Main;

   procedure Dump_On; Dump_Flag := True;
   procedure Dump_Off; Dump_Flag := False;
   procedure Trace_On; Trace_Flag := True;
   procedure Trace_Off; Trace_Flag := False;

   long real procedure Sim_Time;
               Sim_Time := Current.Ev_Time;

   procedure Trace(Message); text Message;
   begin
      if Trace_Flag then
      begin
         OutText(Message&" at");
         OutFix(Sim_Time,2,8);
         OutImage;
      end;
   end;

   ref(proc) procedure Current; Current :- Event_List.First;
```

```
link class proc(Title); text Title;
begin
   long real Ev_Time;
   Boolean Terminated, Failed, Priority;

   procedure Place_in_Event_List(In_Front);
      Boolean In_Front;
   begin
      ref(proc) p;
      if In_Front then Precede(Current) else
      begin
         P :- Event_List.Last;
         while P.Ev_Time>Ev_Time do P:- P.Pred;
         if In_Front then while P.Ev_Time=Ev_Time do P :- P.Pred;
         follow(P);
      end;
      if Dump_Flag then dump_event_list;
   end;

   procedure Wait_Until(Cond, WaitQueue);
         name Cond; Boolean Cond; ref(CondQ) WaitQueue;
   begin
      long real StartTime;
      if not Cond then
      begin
         Failed := True;
         Trace(Title&" waits until");
         Wait(WaitQueue);
         while not Cond do Wait(WaitQueue);
         Trace(Title&" leaves "&WaitQueue.Title);
      end;
      Failed := False;
   end;

   procedure Waken(Delayed); long real Delayed;
   begin
 Trace(Title&" is woken"&Join(" to start at ",Sim_Time+Delayed));
      Out;
      Ev_Time := Sim_Time+Delayed;
      Place_in_Event_List(Priority);
      Resume(Current);
   end;
   Trace(Title&" is created");
   detach;
   inner;
   Trace(Title&" terminates");
   Terminated := True;
   if suc=/= none then Sleep;
end;

procedure Sleep;
begin
   Current.Out;
   if Current =/= none then
   begin
      Trace(Current.Title&" restarts");
      Resume(Current)
   end else Error("Passivate leaves Event List empty");
end;
```

```
   procedure Wait(Q); ref(Head) Q;
   begin
      Current.Into(Q);
      Trace(Current.Title&" restarts");
      Resume(Current);
   end;

   procedure Hold(Delayed); long real Delayed;
   begin
      ref(proc) C;
      C :- Current;
      inspect C do
      begin
       Trace(Title&" holds"&Join("to restart at",Sim_Time+Delayed));
         Ev_Time := Sim_Time + Delayed;
         if suc=/=none and then suc qua Proc.Ev_Time<=Ev_Time then
         begin
            Out;
            Place_in_Event_List(False);
            Trace(Current.Title&" is restarted");
            resume(Current);
         end;
      end;
   end;

   proc class main_proc;
   begin
      while true do detach;
   end;

   head class CondQ(Title); text Title;
   begin
      procedure Signal(Sender); text Sender;
      begin
         Boolean Failed;
         ref(Proc) Next;
         Trace(Title&" is signalled by "&Sender);
         Next :- First;
         Failed := True;
         while Next =/= none and then Failed do
         begin
            Next.Priority := True;
            Next.Waken(0.0);
            Next.Priority := False;
            Failed := Next.Failed;
            Next :- Next.Suc;
         end;
      end;
   end;

   Trace("Simulation starts");
   Event_List :- new Head;
   Main :- new Main_Proc("My Sim");
   Main.Into(Event_List);
   inner;
   Trace("Simulation ends");
end;
```

Chapter 6

EWrap.sim

```
external class DEMOS;
DEMOS class EWRAP;
begin

   Entity class Source(InQ, Rate); ref(Bin)InQ; real Rate;
   begin
      ref(RDist) Arr_T;
      Arr_T :- new NegExp("Arrs",Rate);
      while True do
      begin
         Hold(Arr_T.Sample);
         InQ.Give(1);
      end;
   end;

   ref(RDist) T_Time, BackOff;
   ref (Res) Ether;
   integer I, N_Stations;

   BackOff :- new Uniform("Backoff",0.001,0.5);
   T_Time :- new Uniform("Trans", 0.01,3);
   Ether :- new Res("Ether",1);
   Sysout.OutText("How many stations?"); BreakOutImage;
   N_Stations := Sysin.InInt;
   inner;
   Sysout.OutText("Tracing y/n?"); BreakOutImage;
   InImage; if Sysin.InChar='y' then Trace;
   Sysout.OutText("How long for this run?"); BreakOutImage;
   Hold(Sysin.InReal);

end;
```

Figure 6.12

```
begin
   external class EWrap;
   EWrap
   begin

      entity class Station(InQ); ref(Bin) InQ;
      begin
         while True do
         begin
            InQ.Take(1);
            Ether.AcQuire(1);
           ! Transmit;
            Hold(T_Time.Sample);
            Ether.Release(1);
         end;
      end..of..Transmitter;

      for I := 1 step 1 until N_Stations do
      begin
         ref(Bin) InQ;
         InQ :- new Bin(Edit("InQ",I),0);
         new Source("Source",InQ,0.3).Schedule(0.0);
         new Station("Station",InQ).Schedule(0);
      end;
   end--of--EtherWrap--block;
end++of++program
```

Trace of Figure 6.12

```
  TIME/ CURRENT AND ITS ACTION(S)
0.000 DEMOS    HOLDS FOR 10.000, UNTIL 10.000
   Source 1 HOLDS FOR 2.044, UNTIL 2.044
   Station 1    AWAITS 1 OF InQ 1
   Source 2 HOLDS FOR 1.538, UNTIL 1.538
   Station 2    AWAITS 1 OF InQ 2
   Source 3 HOLDS FOR 1.084, UNTIL 1.084
   Station 3    AWAITS 1 OF InQ 3
 1.084 Source 3 GIVES 1 TO InQ 3
        HOLDS FOR 2.584, UNTIL 3.668
   Station 3    SEIZES 1 OF InQ 3
        SEIZES 1 OF Ether
        HOLDS FOR 2.127, UNTIL 3.211
 1.538 Source 2 GIVES 1 TO InQ 2
        HOLDS FOR 0.077, UNTIL 1.615
   Station 2    SEIZES 1 OF InQ 2
        AWAITS 1 OF Ether
 1.615 Source 2 GIVES 1 TO InQ 2
        HOLDS FOR 4.258, UNTIL 5.873
 2.044 Source 1 GIVES 1 TO InQ 1
        HOLDS FOR 22.592, UNTIL 24.636
   Station 1    SEIZES 1 OF InQ 1
        AWAITS 1 OF Ether
 3.211 Station 3    RELEASES 1 TO Ether
        AWAITS 1 OF InQ 3
   Station 2    SEIZES 1 OF Ether
        HOLDS FOR 0.617, UNTIL 3.828
 3.668 Source 3 GIVES 1 TO InQ 3
        HOLDS FOR 4.792, UNTIL 8.460
   Station 3    SEIZES 1 OF InQ 3
        AWAITS 1 OF Ether
 3.828 Station 2    RELEASES 1 TO Ether
        SEIZES 1 OF InQ 2
        AWAITS 1 OF Ether
   Station 1    SEIZES 1 OF Ether
        HOLDS FOR 2.222, UNTIL 6.050
 5.873 Source 2 GIVES 1 TO InQ 2
        HOLDS FOR 2.316, UNTIL 8.189
 6.050 Station 1    RELEASES 1 TO Ether
        AWAITS 1 OF InQ 1
   Station 3    SEIZES 1 OF Ether
        HOLDS FOR 0.667, UNTIL 6.717
 6.717      RELEASES 1 TO Ether
        AWAITS 1 OF InQ 3
   Station 2    SEIZES 1 OF Ether
        HOLDS FOR 1.398, UNTIL 8.115
 8.115      RELEASES 1 TO Ether
        SEIZES 1 OF InQ 2
        SEIZES 1 OF Ether
        HOLDS FOR 1.753, UNTIL 9.868
 8.189 Source 2 GIVES 1 TO InQ 2
        HOLDS FOR 8.025, UNTIL 16.214
 8.460 Source 3 GIVES 1 TO InQ 3
        HOLDS FOR 2.949, UNTIL 11.409
   Station 3    SEIZES 1 OF InQ 3
        AWAITS 1 OF Ether
 9.868 Station 2    RELEASES 1 TO Ether
        SEIZES 1 OF InQ 2
        AWAITS 1 OF Ether
   Station 3    SEIZES 1 OF Ether
        HOLDS FOR 2.367, UNTIL 12.235
```

Figure 6.14

```
begin
   external class EWRAP;
   EWRAP
   begin

      Entity class Station(InQ); ref(Bin) InQ;
      begin
         while True do
         begin
            InQ.Take(1);
            EtherQ.WaitUntil(Ether.Avail>0);
            while EtherQ.Length>0 do
            begin
               Hold(BackOff.Sample);
            end;
            Ether.Acquire(1);
            Hold(T_Time.Sample);
            Ether.Release(1);
            EtherQ.Signal;
         end;
      end--of--Station;

      ref(CondQ) EtherQ;

      EtherQ :- new CondQ("EtherQ");
      for I := 1 step 1 until N_Stations do
      begin
         ref(Bin) InQ;
         InQ :- new Bin(Edit("InQ",I),0);
         new Source("Source",InQ,0.3).Schedule(0);
         new Station("Station",InQ).Schedule(0);
      end;

   end;
end;
```

Trace from Figure 6.14

```
  Source 1 HOLDS FOR 2.044, UNTIL 2.044
  Station 1    AWAITS 1 OF InQ 1
  Source 2 HOLDS FOR 1.538, UNTIL 1.538
  Station 2    AWAITS 1 OF InQ 2
  Source 3 HOLDS FOR 1.084, UNTIL 1.084
  Station 3    AWAITS 1 OF InQ 3
1.084 Source 3 GIVES 1 TO InQ 3
       HOLDS FOR 2.584, UNTIL 3.668
  Station 3    SEIZES 1 OF InQ 3
       SEIZES 1 OF Ether
       HOLDS FOR 2.127, UNTIL 3.211
1.538 Source 2 GIVES 1 TO InQ 2
       HOLDS FOR 0.077, UNTIL 1.615
  Station 2    SEIZES 1 OF InQ 2
       W'UNTIL IN EtherQ
1.615 Source 2 GIVES 1 TO InQ 2
       HOLDS FOR 4.258, UNTIL 5.873
2.044 Source 1 GIVES 1 TO InQ 1
       HOLDS FOR 22.592, UNTIL 24.636
  Station 1    SEIZES 1 OF InQ 1
       W'UNTIL IN EtherQ
3.211 Station 3    RELEASES 1 TO Ether
       SIGNALS EtherQ
       AWAITS 1 OF InQ 3
  Station 2    LEAVES EtherQ
       HOLDS FOR 0.036, UNTIL 3.247
  Station 1    LEAVES EtherQ
       SEIZES 1 OF Ether
       HOLDS FOR 0.617, UNTIL 3.828
3.247 Station 2    AWAITS 1 OF Ether
3.668 Source 3 GIVES 1 TO InQ 3
       HOLDS FOR 4.792, UNTIL 8.460
  Station 3    SEIZES 1 OF InQ 3
       W'UNTIL IN EtherQ
3.828 Station 1    RELEASES 1 TO Ether
       SIGNALS EtherQ
       AWAITS 1 OF InQ 1
  Station 2    SEIZES 1 OF Ether
       HOLDS FOR 2.222, UNTIL 6.050
5.873 Source 2 GIVES 1 TO InQ 2
       HOLDS FOR 2.316, UNTIL 8.189
6.050 Station 2    RELEASES 1 TO Ether
       SIGNALS EtherQ
       SEIZES 1 OF InQ 2
       HOLDS FOR 0.260, UNTIL 6.310
  Station 3    LEAVES EtherQ
       SEIZES 1 OF Ether
       HOLDS FOR 0.667, UNTIL 6.717
6.310 Station 2    AWAITS 1 OF Ether
6.717 Station 3    RELEASES 1 TO Ether
       SIGNALS EtherQ
       AWAITS 1 OF InQ 3
  Station 2    SEIZES 1 OF Ether
       HOLDS FOR 1.398, UNTIL 8.115
8.115      RELEASES 1 TO Ether
       SIGNALS EtherQ
       SEIZES 1 OF InQ 2
       SEIZES 1 OF Ether
       HOLDS FOR 1.753, UNTIL 9.868
8.189 Source 2 GIVES 1 TO InQ 2
       HOLDS FOR 8.025, UNTIL 16.214      Station 3    SEIZES 1 OF InQ 3
8.460 Source 3 GIVES 1 TO InQ 3           W'UNTIL IN EtherQ
       HOLDS FOR 2.949, UNTIL 11.409      9.868 Station 2    RELEASES 1 TO Ether
```

Figure 6.16

```
begin external class EWRAP;
   EWRAP begin
      Entity class Station(InQ); ref(Bin) InQ;
      begin ref (Res) GotOne;
         GotOne :- new Res("G1",1);
         while True do
         begin
            InQ.Take(1);
            GotOne.Acquire(1);
            while GotOne.Avail=0 do
            begin
               EtherQ.Wait;
               if Collided.Avail=0 then begin
                  Ether.Acquire(1);
                  Hold(T_Time.Sample);
                  GotOne.Release(1);
                  Ether.Release(1);
               end else begin
                  Collided.Take(1);
                  Hold(BackOff.Sample);
               end;
            end;
         end;
      end--of--Station;
      Entity class Ether_c;
      begin ref (Entity) Curr_S;
         while True do
         begin
            Ether.Acquire(1);
            Curr_S :- EtherQ.Coopt;
            if EtherQ.Length>0 then begin
               Collided.Give(1);
            end;
            Ether.Release(1);
            Curr_S.Schedule(0.0);
            if EtherQ.Length=0 then Hold(0);
            while EtherQ.Length>0 do
            begin
               Curr_S :- EtherQ.Coopt;
               Collided.Give(1);
               Curr_S.Schedule(0.0);
            end;
         end;
      end--of--Ether_c;
      ref(Bin) Collided; ref(WaitQ) EtherQ;
      for I:= 1 step 1 until N_Stations do begin ref(Bin) InQ;
         InQ :- new Bin("InQ",0);
         new Source("Source",InQ,0.3).Schedule(0);
         new Station("Station",InQ).Schedule(0);
      end;
      new Ether_c("Ethernet").schedule(0.0);
      Collided :- new Bin("Collisions",0);
      EtherQ :- new WaitQ("EtherQ");
   end;
end;
```

Trace from Figure 6.16

```
  TIME/ CURRENT AND ITS ACTION(S)

0.000 DEMOS    HOLDS FOR 10.000, UNTIL 10.000
  Ethernet 1   SEIZES 1 OF Ether
       WAITS IN EtherQ
  Source 1 HOLDS FOR 2.044, UNTIL 2.044
  Station 1    AWAITS 1 OF InQ
  Source 2 HOLDS FOR 1.538, UNTIL 1.538
  Station 2    AWAITS 1 OF InQ
  Source 3 HOLDS FOR 1.084, UNTIL 1.084
  Station 3    AWAITS 1 OF InQ
1.084 Source 3 GIVES 1 TO InQ
       HOLDS FOR 2.584, UNTIL 3.668
  Station 3    SEIZES 1 OF InQ
       SEIZES 1 OF G1
       WAITS IN EtherQ
  Ethernet 1   COOPTS Station 3 FROM EtherQ
       RELEASES 1 TO Ether
       SCHEDULES Station 3 NOW
       HOLDS FOR 0.000, UNTIL 1.084
  Station 3    SEIZES 1 OF Ether
       HOLDS FOR 2.127, UNTIL 3.211
  Ethernet 1   AWAITS 1 OF Ether
1.538 Source 2 GIVES 1 TO InQ
       HOLDS FOR 0.077, UNTIL 1.615
  Station 2    SEIZES 1 OF InQ
       SEIZES 1 OF G1
       WAITS IN EtherQ
1.615 Source 2 GIVES 1 TO InQ
       HOLDS FOR 4.258, UNTIL 5.873
2.044 Source 1 GIVES 1 TO InQ
       HOLDS FOR 22.592, UNTIL 24.636
  Station 1    SEIZES 1 OF InQ
       SEIZES 1 OF G1
       WAITS IN EtherQ
3.211 Station 3    RELEASES 1 TO G1
       RELEASES 1 TO Ether
       AWAITS 1 OF InQ
  Ethernet 1   SEIZES 1 OF Ether
       COOPTS Station 2 FROM EtherQ
       GIVES 1 TO Collisions
       RELEASES 1 TO Ether
       SCHEDULES Station 2 NOW
       COOPTS Station 1 FROM EtherQ
       GIVES 1 TO Collisions
       SCHEDULES Station 1 NOW
       SEIZES 1 OF Ether
       WAITS IN EtherQ
  Station 2    SEIZES 1 OF Collisions
       HOLDS FOR 0.036, UNTIL 3.247
  Station 1    SEIZES 1 OF Collisions
       HOLDS FOR 0.260, UNTIL 3.471
3.247 Station 2    WAITS IN EtherQ
  Ethernet 1   COOPTS Station 2 FROM EtherQ
       RELEASES 1 TO Ether
       SCHEDULES Station 2 NOW
       HOLDS FOR 0.000, UNTIL 3.247
  Station 2    SEIZES 1 OF Ether
       HOLDS FOR 0.617, UNTIL 3.864
  Ethernet 1   AWAITS 1 OF Ether
3.471 Station 1    WAITS IN EtherQ
3.668 Source 3 GIVES 1 TO InQ
       HOLDS FOR 4.792, UNTIL 8.460
  Station 3    SEIZES 1 OF InQ
       SEIZES 1 OF G1
       WAITS IN EtherQ
3.864 Station 2    RELEASES 1 TO G1
       RELEASES 1 TO Ether
       SEIZES 1 OF InQ
       SEIZES 1 OF G1
       WAITS IN EtherQ
  Ethernet 1   SEIZES 1 OF Ether
       COOPTS Station 1 FROM EtherQ
       GIVES 1 TO Collisions
       RELEASES 1 TO Ether
       SCHEDULES Station 1 NOW
       COOPTS Station 3 FROM EtherQ
       GIVES 1 TO Collisions
       SCHEDULES Station 3 NOW
       COOPTS Station 2 FROM EtherQ
       GIVES 1 TO Collisions
       SCHEDULES Station 2 NOW
       SEIZES 1 OF Ether
       WAITS IN EtherQ
  Station 1    SEIZES 1 OF Collisions
       HOLDS FOR 0.210, UNTIL 4.074
  Station 3    SEIZES 1 OF Collisions
       HOLDS FOR 0.112, UNTIL 3.976
  Station 2    SEIZES 1 OF Collisions
       HOLDS FOR 0.303, UNTIL 4.167
3.976 Station 3    WAITS IN EtherQ
  Ethernet 1   COOPTS Station 3 FROM EtherQ
       RELEASES 1 TO Ether
       SCHEDULES Station 3 NOW
       HOLDS FOR 0.000, UNTIL 3.976
  Station 3    SEIZES 1 OF Ether
       HOLDS FOR 2.222, UNTIL 6.198
  Ethernet 1   AWAITS 1 OF Ether
4.074 Station 1    WAITS IN EtherQ
4.167 Station 2    WAITS IN EtherQ
5.873 Source 2 GIVES 1 TO InQ
       HOLDS FOR 2.316, UNTIL 8.189
6.198 Station 3    RELEASES 1 TO G1
       RELEASES 1 TO Ether
       AWAITS 1 OF InQ
  Ethernet 1   SEIZES 1 OF Ether
       COOPTS Station 1 FROM EtherQ
       GIVES 1 TO Collisions
       RELEASES 1 TO Ether
       SCHEDULES Station 1 NOW
       COOPTS Station 2 FROM EtherQ
       GIVES 1 TO Collisions
       SCHEDULES Station 2 NOW
       SEIZES 1 OF Ether
       WAITS IN EtherQ
  Station 1    SEIZES 1 OF Collisions
       HOLDS FOR 0.158, UNTIL 6.356
  Station 2    SEIZES 1 OF Collisions
       HOLDS FOR 0.324, UNTIL 6.522
6.356 Station 1    WAITS IN EtherQ
  Ethernet 1   COOPTS Station 1 FROM EtherQ
       RELEASES 1 TO Ether
       SCHEDULES Station 1 NOW
       HOLDS FOR 0.000, UNTIL 6.356
  Station 1    SEIZES 1 OF Ether
       HOLDS FOR 0.667, UNTIL 7.023
  Ethernet 1   AWAITS 1 OF Ether
6.522 Station 2    WAITS IN EtherQ
7.023 Station 1    RELEASES 1 TO G1
       RELEASES 1 TO Ether
       AWAITS 1 OF InQ
  Ethernet 1   SEIZES 1 OF Ether
       COOPTS Station 2 FROM EtherQ
       RELEASES 1 TO Ether
       SCHEDULES Station 2 NOW
       HOLDS FOR 0.000, UNTIL 7.023
  Station 2    SEIZES 1 OF Ether
       HOLDS FOR 1.398, UNTIL 8.421
  Ethernet 1   AWAITS 1 OF Ether
8.189 Source 2 GIVES 1 TO InQ
       HOLDS FOR 8.025, UNTIL 16.214
8.421 Station 2    RELEASES 1 TO G1
       RELEASES 1 TO Ether
       SEIZES 1 OF InQ
       SEIZES 1 OF G1
       WAITS IN EtherQ
  Ethernet 1   SEIZES 1 OF Ether
       COOPTS Station 2 FROM EtherQ
       RELEASES 1 TO Ether
       SCHEDULES Station 2 NOW
       HOLDS FOR 0.000, UNTIL 8.421
  Station 2    SEIZES 1 OF Ether
       HOLDS FOR 1.753, UNTIL 10.174
  Ethernet 1   AWAITS 1 OF Ether
8.460 Source 3 GIVES 1 TO InQ
       HOLDS FOR 2.949, UNTIL 11.409
  Station 3    SEIZES 1 OF InQ
       SEIZES 1 OF G1
       WAITS IN EtherQ
```

Figure 6.18

```
Begin
   External Class Demos;
   Demos
     Begin
         Ref(Res) Buffers;
         Real T_read, T_update, T_gather, T_use, T_sim;

         Entity Class Reader;
         Begin
            Buffers.Acquire(1);
            Hold(T_read);              ! Read;
            Buffers.Release(1);
            Hold(T_use);               ! Use;
         End Of Reader;

         Entity Class Writer;
         Begin
            Buffers.Acquire(3);
            Hold(T_update);            ! Update;
            Buffers.Release(3);
            Hold(T_gather);            ! Gather;
         End Of Writer;

         T_read:=Inreal;T_use:=Inreal;
         T_update:=Inreal;T_gather:=Inreal;
         T_sim := Inreal;

         Buffers :- New Res("Buffers",   3);
         New Reader("Reader").Schedule(0.0);
         New Reader("Reader").Schedule(0.0);
         New Writer("Writer").Schedule(0.0);
         Hold(T_sim);
      End;
End
```

Appendix C

This Appendix contains the CCS models of all models in Chapters 3 and 6 of this dissertation and, where appropriate, the corresponding Concurrency Workbench experiments using them.

Chapter 3

Figure 3.2

Model

```
bi P0 3.0

bi P1 2.'esched.$0
bi P2 $esched.1.0

bi P3 (P1 | P2)\{esched}
```

Output

```
Command: states P0
1: 0
2: 1.0
3: 2.0
4: P0

Command: states P3
1: ($0 | 0)\esched
2: ($0 | 1.0)\esched
3: ('esched.$0 | $esched.1.0)\esched
4: (1.'esched.$0 | $esched.1.0)\esched
5: P3

Command: statesobs P3
=== 1 1 1 ===> ($0 | 0)\esched
=== 1 1 ===> ($0 | 1.0)\esched
=== 1 1 ===> ('esched.$0 | $esched.1.0)\esched
=== 1 ===> (1.'esched.$0 | $esched.1.0)\esched
===  ===> P3

Command: statesobs P0
=== 1 1 1 ===> 0
=== 1 1 ===> 1.0
=== 1 ===> 2.0
===  ===> P0

Command: cong
Agent: P0
Agent: P3
true
Command: eq
Agent: P0
Agent: P3
true
```

Figure 3.3

Model

```
bi P0 3.P0

bi P1 $esched1.2.'esched2.P1
bi P2 $esched2.1.'esched1.P2

bi P3 (P1 | 'esched1.P2)\{esched1,esched2}
```

Output

```
Command: if m303.cwb
done.
Command: states P0
1: 1.P0
2: 2.P0
3: P0

Command: states P3
1: ($esched1.2.'esched2.P1 | 'esched1.P2)\{esched1,esched2}
2: (P1 | 1.'esched1.P2)\{esched1,esched2}
3: ('esched2.P1 | $esched2.1.'esched1.P2)\{esched1,esched2}
4: (1.'esched2.P1 | $esched2.1.'esched1.P2)\{esched1,esched2}
5: (2.'esched2.P1 | P2)\{esched1,esched2}
6: P3

Command: eq
Agent: P0
Agent: P3
true
Command: cong
Agent: P0
Agent: P3
false
Command: statesobs P0
=== 1 1 ===> 1.P0
=== 1 ===> 2.P0
===  ===> P0

Command: statesobs P3
=== 1 1 1 ===> ($esched1.2.'esched2.P1 | 'esched1.P2)\{esched1,esched2}
=== 1 1 ===> (P1 | 1.'esched1.P2)\{esched1,esched2}
=== 1 1 ===> ('esched2.P1 | $esched2.1.'esched1.P2)\{esched1,esched2}
=== 1 ===> (1.'esched2.P1 | $esched2.1.'esched1.P2)\{esched1,esched2}
===  ===> (2.'esched2.P1 | P2)\{esched1,esched2}
===  ===> P3
```

Figure 3.4

Model

```
bi Seq $'hammerAcq1.3.'hammerRel1.$0

bi Hammer1 $hammerAcq1.Hammer0
bi Hammer0 $hammerRel1.Hammer1

bi Model (Seq|Hammer1)\{hammerAcq1,hammerRel1}
```

Output

```
Command: reduce

Command: states Model
1: Model
      = ($'hammerAcq1.3.'hammerRel1.$0 | $hammerAcq1.Hammer0)\{hammerAcq1,hammerRel1}
2: (3.'hammerRel1.$0 | Hammer0)\{hammerAcq1,hammerRel1}
3: (2.'hammerRel1.$0 | $hammerRel1.Hammer1)\{hammerAcq1,hammerRel1}
4: (1.'hammerRel1.$0 | $hammerRel1.Hammer1)\{hammerAcq1,hammerRel1}
5: ('hammerRel1.$0 | $hammerRel1.Hammer1)\{hammerAcq1,hammerRel1}
6: ($0 | Hammer1)\{hammerAcq1,hammerRel1}
      = ($0 | $hammerAcq1.Hammer0)\{hammerAcq1,hammerRel1}
```

Figure 3.5

Model

```
bi Seq $'hammerAcq1.3.'hammerRel1.Seq

bi Hammer1 $hammerAcq1.Hammer0
bi Hammer0 $hammerRel1.Hammer1

bi Model (Seq|Hammer1)\{hammerAcq1,hammerRel1}
```

Output

```
Command: states Model
1: Model
      = ($'hammerAcq1.3.'hammerRel1.Seq | $hammerAcq1.Hammer0)\{hammerAcq1,hammerRel1}
      = (Seq | Hammer1)\{hammerAcq1,hammerRel1}
2: (3.'hammerRel1.Seq | Hammer0)\{hammerAcq1,hammerRel1}
3: (2.'hammerRel1.Seq | $hammerRel1.Hammer1)\{hammerAcq1,hammerRel1}
4: (1.'hammerRel1.Seq | $hammerRel1.Hammer1)\{hammerAcq1,hammerRel1}
5: ('hammerRel1.Seq | $hammerRel1.Hammer1)\{hammerAcq1,hammerRel1}
```

Figure 3.6

Model

```
bi Seq1 'valAss4.Seq2
bi Seq2 (valGet4.'valAss6.Seq3+valGet5.'valAss7.Seq3+valGet6.'valAss8.Seq3\
+valGet7.valAss9.Seq3+valGet8.'valAss10.Seq3+valGet9.'valAss11.Seq3\
+valGet10.'valAss12.Seq3)
bi Seq3 3.(valGet4.'valAss8.Seq2+valGet5.'valAss10.Seq2+valGet6.'valAss12.Seq2\
+valGet7.valAss14.Seq2+valGet8.'valAss16.Seq2+valGet9.'valAss18.Seq2\
+valGet10.'valAss20.Seq2)

bi Val0 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+ $'valGet0.Val0
bi Val1 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+ $'valGet1.Val1
bi Val2 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+ $'valGet2.Val2
bi Val3 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+ $'valGet3.Val3
bi Val4 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+ $'valGet4.Val4
bi Val5 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+ $'valGet5.Val5
bi Val6 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+ $'valGet6.Val6
bi Val7 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+ $'valGet7.Val7
bi Val8 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+ $'valGet8.Val8
bi Val9 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+ $'valGet9.Val9
bi Val10 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+ $'valGet10.Val10

bi Seq (Seq1 | Val0)\
\{valAss1,valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,\
valAss8,valAss9,valAss10,valAss11,valAss12,\
valGet0,valGet1,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,\
valGet8,valGet9,valGet10}
```

Output

```
Command: States Seq
1: ('valAss12.Seq2 |
Val6)\{valAss1,valAss10,valAss11,valAss12,valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8
,valAss9,valGet0,valGet1,valGet10,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9
}
2: ((valGet4.'valAss8.Seq2 + valGet5.'valAss10.Seq2 + valGet6.'valAss12.Seq2 +
valGet7.valAss14.Seq2 + valGet8.'valAss16.Seq2 + valGet9.'valAss18.Seq2 +
valGet10.'valAss20.Seq2) |
$'valGet6.Val6)\{valAss1,valAss10,valAss11,valAss12,valAss2,valAss3,valAss4,valAss5,valAss6,valAs
s7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGe
t8,valGet9}
3: (1.(valGet4.'valAss8.Seq2 + valGet5.'valAss10.Seq2 + valGet6.'valAss12.Seq2 +
valGet7.valAss14.Seq2 + valGet8.'valAss16.Seq2 + valGet9.'valAss18.Seq2 +
valGet10.'valAss20.Seq2) |
$'valGet6.Val6)\{valAss1,valAss10,valAss11,valAss12,valAss2,valAss3,valAss4,valAss5,valAss6,valAs
s7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGe
t8,valGet9}
4: (2.(valGet4.'valAss8.Seq2 + valGet5.'valAss10.Seq2 + valGet6.'valAss12.Seq2 +
valGet7.valAss14.Seq2 + valGet8.'valAss16.Seq2 + valGet9.'valAss18.Seq2 +
valGet10.'valAss20.Seq2) |
$'valGet6.Val6)\{valAss1,valAss10,valAss11,valAss12,valAss2,valAss3,valAss4,valAss5,valAss6,valAs
s7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGe
t8,valGet9}
5: (Seq3 |
Val6)\{valAss1,valAss10,valAss11,valAss12,valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8
,valAss9,valGet0,valGet1,valGet10,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9
}
6: ('valAss6.Seq3 |
Val4)\{valAss1,valAss10,valAss11,valAss12,valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8
,valAss9,valGet0,valGet1,valGet10,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9
}
7: (Seq2 |
Val4)\{valAss1,valAss10,valAss11,valAss12,valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8
,valAss9,valGet0,valGet1,valGet10,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9
}
8: Seq

Command: statesobs Seq
=== 1 1 1 ===> ('valAss12.Seq2 |
Val6)\{valAss1,valAss10,valAss11,valAss12,valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8
,valAss9,valGet0,valGet1,valGet10,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9
}
=== 1 1 1 ===> ((valGet4.'valAss8.Seq2 + valGet5.'valAss10.Seq2 + valGet6.'valAss12.Seq2 +
valGet7.valAss14.Seq2 + valGet8.'valAss16.Seq2 + valGet9.'valAss18.Seq2 +
valGet10.'valAss20.Seq2) |
$'valGet6.Val6)\{valAss1,valAss10,valAss11,valAss12,valAss2,valAss3,valAss4,valAss5,valAss6,valAs
s7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGe
t8,valGet9}
=== 1 1 ===> (1.(valGet4.'valAss8.Seq2 + valGet5.'valAss10.Seq2 + valGet6.'valAss12.Seq2 +
valGet7.valAss14.Seq2 + valGet8.'valAss16.Seq2 + valGet9.'valAss18.Seq2 +
valGet10.'valAss20.Seq2) |
$'valGet6.Val6)\{valAss1,valAss10,valAss11,valAss12,valAss2,valAss3,valAss4,valAss5,valAss6,valAs
s7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGe
t8,valGet9}
=== 1 ===> (2.(valGet4.'valAss8.Seq2 + valGet5.'valAss10.Seq2 + valGet6.'valAss12.Seq2 +
valGet7.valAss14.Seq2 + valGet8.'valAss16.Seq2 + valGet9.'valAss18.Seq2 +
valGet10.'valAss20.Seq2) |
$'valGet6.Val6)\{valAss1,valAss10,valAss11,valAss12,valAss2,valAss3,valAss4,valAss5,valAss6,valAs
s7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGe
t8,valGet9}
===  ===> (Seq3 |
Val6)\{valAss1,valAss10,valAss11,valAss12,valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8
,valAss9,valGet0,valGet1,valGet10,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9
}
===  ===> ('valAss6.Seq3 |
Val4)\{valAss1,valAss10,valAss11,valAss12,valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8
,valAss9,valGet0,valGet1,valGet10,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9
}
===  ===> (Seq2 |
Val4)\{valAss1,valAss10,valAss11,valAss12,valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8
,valAss9,valGet0,valGet1,valGet10,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9
}
===  ===> Seq
```

Figure 3.7

Model

```
bi Seq1 'valAss4.Seq2

bi Seq2 (valGet4.'valAss6.Seq3+valGet5.'valAss7.Seq3+valGet6.'valAss8.Seq3\
+valGet7.valAss9.Seq3+valGet8.'valAss10.Seq3+valGet9.'valAss11.Seq3\
+valGet10.'valAss12.Seq3+valGet11.'valAss13.Seq3+valGet12.'valAss14.Seq3)
bi Seq3 3.(valGet4.'valAss8.Seq2+valGet5.'valAss10.Seq2+valGet6.'valAss12.Seq2\
+valGet7.valAss14.Seq2+valGet8.'valAss16.Seq2+valGet9.'valAss18.Seq2\
+valGet10.'valAss4.Seq2+valGet11.'valAss4.Seq2+valGet12.'valAss4.Seq2\
+valGet13.'valAss4.Seq2+valGet14.'valAss4.Seq2)
bi Val0 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet0.Val0
bi Val1 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet1.Val1
bi Val2 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet2.Val2
bi Val3 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet3.Val3
bi Val4 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet4.Val4
bi Val5 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet5.Val5
bi Val6 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet6.Val6
bi Val7 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet7.Val7
bi Val8 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet8.Val8
bi Val9 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet9.Val9
bi Val10 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet10.Val10
bi Val11 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet11.Val11
bi Val12 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet12.Val12
bi Val13 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet13.Val13
bi Val14 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet14.Val14

bi Seq (Seq1 | Val0)\
\{valAss1,valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,\
valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,\
valGet0,valGet1,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,\
valGet8,valGet9,valGet10,valGet11,valGet12,valGet13,valGet14}
```

Output

```
Command: states Seq
1: ('valAss4.Seq2 |
Val14)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,valAss2,v
alAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11,valGet12
,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
2: ((valGet4.'valAss8.Seq2 + valGet5.'valAss10.Seq2 + valGet6.'valAss12.Seq2 +
valGet7.valAss14.Seq2 + valGet8.'valAss16.Seq2 + valGet9.'valAss18.Seq2 + valGet10.'valAss4.Seq2
+ valGet11.'valAss4.Seq2 + valGet12.'valAss4.Seq2 + valGet13.'valAss4.Seq2 +
valGet14.'valAss4.Seq2) |
$'valGet14.Val14)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss1
8,valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet
11,valGet12,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
3: (1.(valGet4.'valAss8.Seq2 + valGet5.'valAss10.Seq2 + valGet6.'valAss12.Seq2 +
valGet7.valAss14.Seq2 + valGet8.'valAss16.Seq2 + valGet9.'valAss18.Seq2 + valGet10.'valAss4.Seq2
+ valGet11.'valAss4.Seq2 + valGet12.'valAss4.Seq2 + valGet13.'valAss4.Seq2 +
valGet14.'valAss4.Seq2) |
$'valGet14.Val14)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss1
8,valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet
11,valGet12,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
4: (2.(valGet4.'valAss8.Seq2 + valGet5.'valAss10.Seq2 + valGet6.'valAss12.Seq2 +
valGet7.valAss14.Seq2 + valGet8.'valAss16.Seq2 + valGet9.'valAss18.Seq2 + valGet10.'valAss4.Seq2
+ valGet11.'valAss4.Seq2 + valGet12.'valAss4.Seq2 + valGet13.'valAss4.Seq2 +
valGet14.'valAss4.Seq2) |
$'valGet14.Val14)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss1
8,valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet
11,valGet12,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
5: (Seq3 |
Val14)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,valAss2,v
alAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11,valGet12
,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
6: ('valAss14.Seq3 |
Val12)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,valAss2,v
alAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11,valGet12
,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
7: (Seq2 |
Val12)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,valAss2,v
alAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11,valGet12
,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
8: ('valAss12.Seq2 |
Val6)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,valAss2,va
lAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11,valGet12,
valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
9: ((valGet4.'valAss8.Seq2 + valGet5.'valAss10.Seq2 + valGet6.'valAss12.Seq2 +
valGet7.valAss14.Seq2 + valGet8.'valAss16.Seq2 + valGet9.'valAss18.Seq2 + valGet10.'valAss4.Seq2
+ valGet11.'valAss4.Seq2 + valGet12.'valAss4.Seq2 + valGet13.'valAss4.Seq2 +
valGet14.'valAss4.Seq2) |
$'valGet6.Val6)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,
valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11
,valGet12,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
10: (1.(valGet4.'valAss8.Seq2 + valGet5.'valAss10.Seq2 + valGet6.'valAss12.Seq2 +
valGet7.valAss14.Seq2 + valGet8.'valAss16.Seq2 + valGet9.'valAss18.Seq2 + valGet10.'valAss4.Seq2
+ valGet11.'valAss4.Seq2 + valGet12.'valAss4.Seq2 + valGet13.'valAss4.Seq2 +
valGet14.'valAss4.Seq2) |
$'valGet6.Val6)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,
valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11
,valGet12,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
11: (2.(valGet4.'valAss8.Seq2 + valGet5.'valAss10.Seq2 + valGet6.'valAss12.Seq2 +
valGet7.valAss14.Seq2 + valGet8.'valAss16.Seq2 + valGet9.'valAss18.Seq2 + valGet10.'valAss4.Seq2
+ valGet11.'valAss4.Seq2 + valGet12.'valAss4.Seq2 + valGet13.'valAss4.Seq2 +
valGet14.'valAss4.Seq2) |
$'valGet6.Val6)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,
valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11
,valGet12,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
12: (Seq3 |
Val6)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,valAss2,va
lAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11,valGet12,
valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
13: ('valAss6.Seq3 |
Val4)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,valAss2,va
lAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11,valGet12,
valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
14: (Seq2 |
Val4)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,valAss2,va
lAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11,valGet12,
valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
15: Seq
```

Figure 3.8

Model

```
bi Seq1 'valAss4.Seq2

bi Seq2 (valGet4.'valAss6.Seq3+valGet5.'valAss7.Seq3+valGet6.'valAss8.Seq3\
+valGet7.valAss9.Seq3+valGet8.'valAss10.Seq3+valGet9.'valAss11.Seq3\
+valGet10.$0+valGet11.$0+valGet12.$0)

bi Seq3 3.Seq2

bi Val0 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet0.Val0

bi Val1 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet1.Val1

bi Val2 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet2.Val2

bi Val3 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet3.Val3

bi Val4 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet4.Val4

bi Val5 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet5.Val5
bi Val6 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet6.Val6
bi Val7 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet7.Val7
bi Val8 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet8.Val8
bi Val9 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet9.Val9
bi Val10 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet10.Val10
bi Val11 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet11.Val11
bi Val12 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet12.Val12
bi Val13 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet13.Val13
bi Val14 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4\
+valAss5.Val5+valAss6.Val6+valAss7.Val7+valAss8.Val8\
+valAss9.Val9+valAss10.Val10+valAss11.Val11+valAss12.Val12+\
valAss13.Val13+valAss14.Val14+ $'valGet14.Val14

bi Seq (Seq1 | Val0)\
\{valAss1,valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,\
valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,\
valGet0,valGet1,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,\
valGet8,valGet9,valGet10,valGet11,valGet12,valGet13,valGet14}
```

Output

```
Command: states Seq
1: ($0 |
$'valGet10.Val10)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss1
8,valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet
11,valGet12,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
2: ($0 |
Val10)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,valAss2,v
alAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11,valGet12
,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
3: (Seq2 |
$'valGet10.Val10)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss1
8,valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet
11,valGet12,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
4: (1.Seq2 |
$'valGet10.Val10)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss1
8,valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet
11,valGet12,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
5: (2.Seq2 |
$'valGet10.Val10)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss1
8,valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet
11,valGet12,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
6: (Seq3 |
Val10)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,valAss2,v
alAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11,valGet12
,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
7: ('valAss10.Seq3 |
Val8)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,valAss2,va
lAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11,valGet12,
valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
8: (Seq2 |
$'valGet8.Val8)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,
valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11
,valGet12,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
9: (1.Seq2 |
$'valGet8.Val8)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,
valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11
,valGet12,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
10: (2.Seq2 |
$'valGet8.Val8)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,
valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11
,valGet12,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
11: (Seq3 |
Val8)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,valAss2,va
lAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11,valGet12,
valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
12: ('valAss8.Seq3 |
Val6)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,valAss2,va
lAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11,valGet12,
valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
13: (Seq2 |
$'valGet6.Val6)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,
valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11
,valGet12,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
14: (1.Seq2 |
$'valGet6.Val6)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,
valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11
,valGet12,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
15: (2.Seq2 |
$'valGet6.Val6)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,
valAss2,valAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11
,valGet12,valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
16: (Seq3 |
Val6)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,valAss2,va
lAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11,valGet12,
valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
17: ('valAss6.Seq3 |
Val4)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,valAss2,va
lAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11,valGet12,
valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
18: (Seq2 |
Val4)\{valAss1,valAss10,valAss11,valAss12,valAss13,valAss14,valAss16,valAss17,valAss18,valAss2,va
lAss3,valAss4,valAss5,valAss6,valAss7,valAss8,valAss9,valGet0,valGet1,valGet10,valGet11,valGet12,
valGet13,valGet14,valGet2,valGet3,valGet4,valGet5,valGet6,valGet7,valGet8,valGet9}
19: Seq
```

Figure 3.9

Model

```
bi Station (3.Packet | 2.Station)
bi Packet $0
```

Output

```
Command: sim Station

Simulated agent: Station
Transitions:
    1: --- 1 ---> 2.Packet | 1.Station

Sim> 1
  --- 1 --->

Simulated agent: 2.Packet | 1.Station
Transitions:
    1: --- 1 ---> 1.Packet | Station

Sim> 1
  --- 1 --->

Simulated agent: 1.Packet | Station
Transitions:
    1: --- 1 ---> Packet | (2.Packet | 1.Station)

Sim> 1
  --- 1 --->

Simulated agent: Packet | (2.Packet | 1.Station)
Transitions:
    1: --- 1 ---> $0 | (1.Packet | Station)

Sim> 1
  --- 1 --->

Simulated agent: $0 | (1.Packet | Station)
Transitions:
    1: --- 1 ---> $0 | (Packet | (2.Packet | 1.Station))

Sim> 1
  --- 1 --->

Simulated agent: $0 | (Packet | (2.Packet | 1.Station))
Transitions:
    1: --- 1 ---> $0 | ($0 | (1.Packet | Station))

Sim> 1
  --- 1 --->

Simulated agent: $0 | ($0 | (1.Packet | Station))
Transitions:
    1: --- 1 ---> $0 | ($0 | (Packet | (2.Packet | 1.Station)))
```

Figure 3.10

Model

```
bi Station (3.'pSched.$0 | 2.Station)
bi Packet $psched.$0

Output

Command: sim Station

Simulated agent: Station
Transitions:
    1: --- 1 ---> 2.'pSched.$0 | 1.Station

Sim> 1
  --- 1 --->

Simulated agent: 2.'pSched.$0 | 1.Station
Transitions:
    1: --- 1 ---> 1.'pSched.$0 | Station

Sim> 1
  --- 1 --->

Simulated agent: 1.'pSched.$0 | Station
Transitions:
    1: --- 1 ---> 'pSched.$0 | (2.'pSched.$0 | 1.Station)

Sim> 1
  --- 1 --->

Simulated agent: 'pSched.$0 | (2.'pSched.$0 | 1.Station)
Transitions:
    1: --- 'pSched ---> $0 | (2.'pSched.$0 | 1.Station)

Sim> 1
  --- 'pSched --->

Simulated agent: $0 | (2.'pSched.$0 | 1.Station)
Transitions:
    1: --- 1 ---> $0 | (1.'pSched.$0 | Station)

Sim> 1
  --- 1 --->

Simulated agent: $0 | (1.'pSched.$0 | Station)
Transitions:
    1: --- 1 ---> $0 | ('pSched.$0 | (2.'pSched.$0 | 1.Station))

Sim> 1
  --- 1 --->

Simulated agent: $0 | ('pSched.$0 | (2.'pSched.$0 | 1.Station))
Transitions:
    1: --- 'pSched ---> $0 | ($0 | (2.'pSched.$0 | 1.Station))
```

Figure 3.11

Model

```
bi Boat ($'jacq1.$'tugacq2.1.'tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0\
| 2.Boat)

bi Tugs3 $tugacq1.Tugs2 + $tugacq2.Tugs1 + $tugacq3.Tugs0
bi Tugs2 $tugacq1.Tugs1 + $tugacq2.Tugs0 + $tugrel1.Tugs3
bi Tugs1 $tugacq1.Tugs0 + $tugrel1.Tugs2 + $tugrel2.Tugs3
bi Tugs0 $tugrel1.Tugs1 + $tugrel2.Tugs2 + $tugrel3.Tugs3

bi Jetties2 $jacq1.Jetties1 + $jacq2.Jetties0
bi Jetties1 $jacq1.Jetties0 + $jrel1.Jetties2
bi Jetties0 $jrel1.Jetties1 + $jrel2.Jetties2

bi Model (Boat | Tugs3 | Jetties2)\{tugacq1,tugacq2,tugacq3,\
tugrel1,tugrel2,tugrel3,jacq1,jacq2,jrel1,jrel2}
```

Output

```
Command: sim Model
Simulated agent: Model
Transitions:
    1: --- t<jacq1> ---> (($'tugacq2.1.'tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 2.Boat) |
Tugs3 | Jetties1)\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
    2: --- 1 ---> (($'jacq1.$'tugacq2.1.'tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 1.Boat) |
($tugacq1.Tugs2 + $tugacq2.Tugs1 + $tugacq3.Tugs0) | ($jacq1.Jetties1 +
$jacq2.Jetties0))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Sim> 1
  --- t<jacq1> --->
Simulated agent: (($'tugacq2.1.'tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 2.Boat) | Tugs3 |
Jetties1)\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Transitions:
    1: --- t<tugacq2> ---> ((1.'tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 2.Boat) | Tugs1 |
Jetties1)\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
    2: --- 1 ---> (($'tugacq2.1.'tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 1.Boat) |
($tugacq1.Tugs2 + $tugacq2.Tugs1 + $tugacq3.Tugs0) | ($jacq1.Jetties0 +
$jrel1.Jetties2))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Sim> 1
  --- t<tugacq2> --->
Simulated agent: ((1.'tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 2.Boat) | Tugs1 |
Jetties1)\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Transitions:
    1: --- 1 ---> (('tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 1.Boat) | ($tugacq1.Tugs0 +
$tugrel1.Tugs2 + $tugrel2.Tugs3) | ($jacq1.Jetties0 +
$jrel1.Jetties2))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Sim> 1
  --- 1 --->
Simulated agent: (('tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 1.Boat) | ($tugacq1.Tugs0 +
$tugrel1.Tugs2 + $tugrel2.Tugs3) | ($jacq1.Jetties0 +
$jrel1.Jetties2))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Transitions:
    1: --- t<tugrel2> ---> ((1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 1.Boat) | Tugs3 |
($jacq1.Jetties0 +
$jrel1.Jetties2))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Sim> 1
  --- t<tugrel2> --->
Simulated agent: ((1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 1.Boat) | Tugs3 | ($jacq1.Jetties0 +
$jrel1.Jetties2))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Transitions:
    1: --- 1 ---> (($'tugacq1.1.'tugrel1.'jrel1.$0 | Boat) | ($tugacq1.Tugs2 + $tugacq2.Tugs1 +
$tugacq3.Tugs0) | ($jacq1.Jetties0 +
$jrel1.Jetties2))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Sim> 1
  --- 1 --->
Simulated agent: (($'tugacq1.1.'tugrel1.'jrel1.$0 | Boat) | ($tugacq1.Tugs2 + $tugacq2.Tugs1 +
$tugacq3.Tugs0) | ($jacq1.Jetties0 +
$jrel1.Jetties2))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Transitions:
    1: --- t<jacq1> ---> (($'tugacq1.1.'tugrel1.'jrel1.$0 |
($'tugacq2.1.'tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 2.Boat)) | ($tugacq1.Tugs2 +
$tugacq2.Tugs1 + $tugacq3.Tugs0) |
Jetties0)\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
    2: --- t<tugacq1> ---> ((1.'tugrel1.'jrel1.$0 | Boat) | Tugs2 | ($jacq1.Jetties0 +
$jrel1.Jetties2))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
    3: --- 1 ---> (($'tugacq1.1.'tugrel1.'jrel1.$0 |
($'jacq1.$'tugacq2.1.'tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 1.Boat)) | ($tugacq1.Tugs2 +
$tugacq2.Tugs1 + $tugacq3.Tugs0) | ($jacq1.Jetties0 +
$jrel1.Jetties2))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}

Sim> 1
  --- t<jacq1> --->
```

```
Simulated agent: (($'tugacq1.1.'tugrel1.'jrel1.$0 |
($'tugacq2.1.'tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 2.Boat)) | ($tugacq1.Tugs2 +
$tugacq2.Tugs1 + $tugacq3.Tugs0) |
Jetties0)\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Transitions:
    1: --- t<tugacq2> ---> (($'tugacq1.1.'tugrel1.'jrel1.$0 |
(1.'tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 2.Boat)) | Tugs1 |
Jetties0)\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
    2: --- t<tugacq1> ---> ((1.'tugrel1.'jrel1.$0 |
($'tugacq2.1.'tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 2.Boat)) | Tugs2 |
Jetties0)\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
    3: --- 1 ---> (($'tugacq1.1.'tugrel1.'jrel1.$0 |
($'tugacq2.1.'tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 1.Boat)) | ($tugacq1.Tugs2 +
$tugacq2.Tugs1 + $tugacq3.Tugs0) | ($jrel1.Jetties1 +
$jrel2.Jetties2))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Sim> 1
  --- t<tugacq2> --->
Simulated agent: (($'tugacq1.1.'tugrel1.'jrel1.$0 | (1.'tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0
| 2.Boat)) | Tugs1 |
Jetties0)\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Transitions:
    1: --- t<tugacq1> ---> ((1.'tugrel1.'jrel1.$0 | (1.'tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0
| 2.Boat)) | Tugs0 |
Jetties0)\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
    2: --- 1 ---> (($'tugacq1.1.'tugrel1.'jrel1.$0 | ('tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 |
1.Boat)) | ($tugacq1.Tugs0 + $tugrel1.Tugs2 + $tugrel2.Tugs3) | ($jrel1.Jetties1 +
$jrel2.Jetties2))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Sim> 1
  --- t<tugacq1> --->
Simulated agent: ((1.'tugrel1.'jrel1.$0 | (1.'tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 2.Boat))
| Tugs0 | Jetties0)\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Transitions:
    1: --- 1 ---> (('tugrel1.'jrel1.$0 | ('tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 1.Boat)) |
($tugrel1.Tugs1 + $tugrel2.Tugs2 + $tugrel3.Tugs3) | ($jrel1.Jetties1 +
$jrel2.Jetties2))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Sim> 1
  --- 1 --->
Simulated agent: (('tugrel1.'jrel1.$0 | ('tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 1.Boat)) |
($tugrel1.Tugs1 + $tugrel2.Tugs2 + $tugrel3.Tugs3) | ($jrel1.Jetties1 +
$jrel2.Jetties2))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Transitions:
    1: --- t<tugrel1> ---> (('jrel1.$0 | ('tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 1.Boat)) |
Tugs1 | ($jrel1.Jetties1 +
$jrel2.Jetties2))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
    2: --- t<tugrel2> ---> (('tugrel1.'jrel1.$0 | (1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 1.Boat)) |
Tugs2 | ($jrel1.Jetties1 +
$jrel2.Jetties2))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Sim> 1
  --- t<tugrel1> --->
Simulated agent: (('jrel1.$0 | ('tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 1.Boat)) | Tugs1 |
($jrel1.Jetties1 +
$jrel2.Jetties2))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Transitions:
    1: --- t<jrel1> ---> (($0 | ('tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 1.Boat)) | Tugs1 |
Jetties1)\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
    2: --- t<tugrel2> ---> (('jrel1.$0 | (1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 1.Boat)) | Tugs3 |
($jrel1.Jetties1 +
$jrel2.Jetties2))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Sim> 1
  --- t<jrel1> --->
Simulated agent: (($0 | ('tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 1.Boat)) | Tugs1 |
Jetties1)\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Transitions:
    1: --- t<tugrel2> ---> (($0 | (1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 1.Boat)) | Tugs3 |
Jetties1)\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Sim> 1
  --- t<tugrel2> --->
Simulated agent: (($0 | (1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 1.Boat)) | Tugs3 |
Jetties1)\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Transitions:
    1: --- 1 ---> (($0 | ($'tugacq1.1.'tugrel1.'jrel1.$0 | Boat)) | ($tugacq1.Tugs2 +
$tugacq2.Tugs1 + $tugacq3.Tugs0) | ($jacq1.Jetties0 +
$jrel1.Jetties2))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Sim> 1
  --- 1 --->
Simulated agent: (($0 | ($'tugacq1.1.'tugrel1.'jrel1.$0 | Boat)) | ($tugacq1.Tugs2 +
$tugacq2.Tugs1 + $tugacq3.Tugs0) | ($jacq1.Jetties0 +
$jrel1.Jetties2))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
Transitions:
    1: --- t<jacq1> ---> (($0 | ($'tugacq1.1.'tugrel1.'jrel1.$0 |
($'tugacq2.1.'tugrel2.1.$'tugacq1.1.'tugrel1.'jrel1.$0 | 2.Boat))) | ($tugacq1.Tugs2 +
$tugacq2.Tugs1 + $tugacq3.Tugs0) |
Jetties0)\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
    2: --- t<tugacq1> ---> (($0 | (1.'tugrel1.'jrel1.$0 | Boat)) | Tugs2 | ($jacq1.Jetties0 +
$jrel1.Jetties2))\{jacq1,jacq2,jrel1,jrel2,tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3}
```

Figure 3.13 and 3.15

Model

```
bi Producer 2.'widGive1.Producer
bi Consumer $'widTake1.1.Consumer

bi Wid0 $widGive1.Wid1
bi Wid1 $widGive1.Wid2 + $widTake1.Wid0
bi Wid2 $widGive1.Wid2 + $widTake1.Wid1
bi Wid3 $widGive1.Wid4 + $widTake1.Wid2
bi Wid4 $widTake1.Wid3

bi Model (Producer | Consumer | Wid0)\{widTake1,widGive1}
```

Output

```
Command: statesobs Model
===  ===> Model
=== 1 ===> (1.'widGive1.Producer | $'widTake1.1.Consumer | $widGive1.Wid1)\{widGive1,widTake1}
=== 1 1 ===> ('widGive1.Producer | $'widTake1.1.Consumer | $widGive1.Wid1)\{widGive1,widTake1}
=== 1 1 ===> (Producer | $'widTake1.1.Consumer | Wid1)\{widGive1,widTake1}
=== 1 1 ===> (Producer | 1.Consumer | Wid0)\{widGive1,widTake1}
=== 1 1 1 ===> (1.'widGive1.Producer | $'widTake1.1.Consumer | ($widGive1.Wid2 +
$widTake1.Wid0))\{widGive1,widTake1}
=== 1 1 1 ===> (1.'widGive1.Producer | 1.Consumer | Wid0)\{widGive1,widTake1}
=== 1 1 1 1 ===> ('widGive1.Producer | $'widTake1.1.Consumer | ($widGive1.Wid2 +
$widTake1.Wid0))\{widGive1,widTake1}
=== 1 1 1 1 ===> ('widGive1.Producer | Consumer | $widGive1.Wid1)\{widGive1,widTake1}
=== 1 1 1 1 ===> ('widGive1.Producer | 1.Consumer | Wid0)\{widGive1,widTake1}
=== 1 1 1 1 ===> (Producer | $'widTake1.1.Consumer | Wid2)\{widGive1,widTake1}
=== 1 1 1 1 ===> (Producer | 1.Consumer | Wid1)\{widGive1,widTake1}
=== 1 1 1 1 1 ===> (1.'widGive1.Producer | $'widTake1.1.Consumer |
$widTake1.Wid1)\{widGive1,widTake1}
=== 1 1 1 1 1 1 ===> ('widGive1.Producer | $'widTake1.1.Consumer |
$widTake1.Wid1)\{widGive1,widTake1}
=== 1 1 1 1 1 1 ===> ('widGive1.Producer | Consumer | ($widGive1.Wid2 +
$widTake1.Wid0))\{widGive1,widTake1}
=== 1 1 1 1 1 1 ===> ('widGive1.Producer | 1.Consumer | Wid1)\{widGive1,widTake1}
=== 1 1 1 1 1 1 ===> (Producer | 1.Consumer | Wid2)\{widGive1,widTake1}
```

Figure 3.17

Model

```
bi Ferry $cooptFQ1.1.'sched1.1.Ferry + $cooptFQ2.1.'sched2.1.Ferry \
+ $cooptFQ3.1.'sched3.1.Ferry
bi Car1 1.'waitFQ1.$sched1.$0
bi Car2 2.'waitFQ2.$sched2.$0
bi Car3 3.'waitFQ3.$sched3.$0

bi FQ $waitFQ1.FQ1 + $waitFQ2.FQ2 + $waitFQ3.FQ3
bi FQ1 $'cooptFQ1.FQ + $waitFQ2.FQ12 + $waitFQ3.FQ13
bi FQ2 $'cooptFQ2.FQ + $waitFQ3.FQ23 + $waitFQ1.FQ21
bi FQ3 $'cooptFQ3.FQ + $waitFQ1.FQ31 + $waitFQ2.FQ32
bi FQ12  $'cooptFQ1.FQ2 + $waitFQ3.FQ123
bi FQ13  $'cooptFQ1.FQ3 + $waitFQ2.FQ132
bi FQ21  $'cooptFQ2.FQ1 + $waitFQ3.FQ213
bi FQ23  $'cooptFQ2.FQ3 + $waitFQ1.FQ231
bi FQ31  $'cooptFQ3.FQ1 + $waitFQ2.FQ312
bi FQ32  $'cooptFQ3.FQ2 + $waitFQ1.FQ321
bi FQ123 $'cooptFQ1.FQ23
bi FQ132 $'cooptFQ1.FQ32
bi FQ213 $'cooptFQ2.FQ13
bi FQ231 $'cooptFQ2.FQ31
bi FQ312 $'cooptFQ3.FQ12
bi FQ321 $'cooptFQ3.FQ21

bi Model ( Ferry | Car1 | Car2 | Car3 | FQ )\{waitFQ1,waitFQ2,waitFQ3,\
cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3}
```

Output

```
Command: states Model
1: Model
2: (($cooptFQ1.1.'sched1.1.Ferry + $cooptFQ2.1.'sched2.1.Ferry + $cooptFQ3.1.'sched3.1.Ferry) |
'waitFQ1.$sched1.$0 | 1.'waitFQ2.$sched2.$0 | 2.'waitFQ3.$sched3.$0 | ($waitFQ1.FQ1 +
$waitFQ2.FQ2 +
$waitFQ3.FQ3))\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
3: (($cooptFQ1.1.'sched1.1.Ferry + $cooptFQ2.1.'sched2.1.Ferry + $cooptFQ3.1.'sched3.1.Ferry) |
$sched1.$0 | 1.'waitFQ2.$sched2.$0 | 2.'waitFQ3.$sched3.$0 |
FQ1)\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
4: (1.'sched1.1.Ferry | $sched1.$0 | 1.'waitFQ2.$sched2.$0 | 2.'waitFQ3.$sched3.$0 |
FQ)\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
5: (($cooptFQ1.1.'sched1.1.Ferry + $cooptFQ2.1.'sched2.1.Ferry + $cooptFQ3.1.'sched3.1.Ferry) |
$sched1.$0 | 'waitFQ2.$sched2.$0 | 1.'waitFQ3.$sched3.$0 | ($waitFQ2.FQ12 + $waitFQ3.FQ13 +
$'cooptFQ1.FQ)) \{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
6: ('sched1.1.Ferry | $sched1.$0 | 'waitFQ2.$sched2.$0 | 1.'waitFQ3.$sched3.$0 | ($waitFQ1.FQ1 +
$waitFQ2.FQ2 + $waitFQ3.FQ3))
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
7: (1.'sched1.1.Ferry | $sched1.$0 | 'waitFQ2.$sched2.$0 | 1.'waitFQ3.$sched3.$0 | FQ)
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
8: (($cooptFQ1.1.'sched1.1.Ferry + $cooptFQ2.1.'sched2.1.Ferry + $cooptFQ3.1.'sched3.1.Ferry) |
$sched1.$0 | $sched2.$0 | 1.'waitFQ3.$sched3.$0 | FQ12)
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
9: ('sched1.1.Ferry | $sched1.$0 | $sched2.$0 | 1.'waitFQ3.$sched3.$0 | FQ2)
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
      = (1.Ferry | $0 | 'waitFQ2.$sched2.$0 | 1.'waitFQ3.$sched3.$0 | ($waitFQ1.FQ1 +
$waitFQ2.FQ2 + $waitFQ3.FQ3))
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
10: (1.'sched1.1.Ferry | $sched1.$0 | $sched2.$0 | 1.'waitFQ3.$sched3.$0| FQ2)
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
11: (($cooptFQ1.1.'sched1.1.Ferry + $cooptFQ2.1.'sched2.1.Ferry + $cooptFQ3.1.'sched3.1.Ferry) |
$sched1.$0 | $sched2.$0 | 'waitFQ3.$sched3.$0 | ($waitFQ3.FQ123 + $'cooptFQ1.FQ2))
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
12: (1.Ferry | $0 | $sched2.$0 | 1.'waitFQ3.$sched3.$0 | FQ2)
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
13: ('sched1.1.Ferry | $sched1.$0 | $sched2.$0 | 'waitFQ3.$sched3.$0 | ($waitFQ1.FQ21 +
$waitFQ3.FQ23 + $'cooptFQ2.FQ))
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
14: (1.'sched1.1.Ferry | $sched1.$0 | $sched2.$0 | 'waitFQ3.$sched3.$0 | FQ2)
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
15: (($cooptFQ1.1.'sched1.1.Ferry + $cooptFQ2.1.'sched2.1.Ferry + $cooptFQ3.1.'sched3.1.Ferry) |
$sched1.$0 | $sched2.$0 | $sched3.$0 | FQ123)
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
      = (($cooptFQ1.1.'sched1.1.Ferry + $cooptFQ2.1.'sched2.1.Ferry +
$cooptFQ3.1.'sched3.1.Ferry) | $sched1.$0 | $sched2.$0 | $sched3.$0 | $'cooptFQ1.FQ23)
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
16: (Ferry | $0 | $sched2.$0 | 'waitFQ3.$sched3.$0 | ($waitFQ1.FQ21 + $waitFQ3.FQ23 +
$'cooptFQ2.FQ)) \{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
17: ('sched1.1.Ferry | $sched1.$0 | $sched2.$0 | $sched3.$0 | FQ23)
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
      = (1.Ferry | $0 | $sched2.$0 | 'waitFQ3.$sched3.$0 | ($waitFQ1.FQ21 + $waitFQ3.FQ23 +
$'cooptFQ2.FQ)) \{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
18: (1.'sched1.1.Ferry | $sched1.$0 | $sched2.$0 | $sched3.$0 | FQ23)
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
19: (1.'sched2.1.Ferry | $0 | $sched2.$0 | 'waitFQ3.$sched3.$0 |
FQ)\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
20: (Ferry | $0 | $sched2.$0 | $sched3.$0 | FQ23)
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
      = (($cooptFQ1.1.'sched1.1.Ferry + $cooptFQ2.1.'sched2.1.Ferry +
$cooptFQ3.1.'sched3.1.Ferry) | $0 | $sched2.$0 | $sched3.$0 | ($waitFQ1.FQ231 + $'cooptFQ2.FQ3))
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
  = (Ferry | $0 | $sched2.$0 | $sched3.$0 | ($waitFQ1.FQ231 + $'cooptFQ2.FQ3))
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
21: (1.Ferry | $0 | $sched2.$0 | $sched3.$0 | FQ23)
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
      = (1.Ferry | $0 | $sched2.$0 | $sched3.$0 | ($waitFQ1.FQ231 + $'cooptFQ2.FQ3))
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
22: ('sched1.1.Ferry | $sched1.$0 | $sched2.$0 | $sched3.$0 | ($waitFQ1.FQ231 + $'cooptFQ2.FQ3))
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
23: (1.'sched2.1.Ferry | $0 | $sched2.$0 | $sched3.$0 | FQ3)
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
24: ('sched2.1.Ferry | $0 | $sched2.$0 | $sched3.$0 | ($waitFQ1.FQ31 + $waitFQ2.FQ32 +
$'cooptFQ3.FQ)) \{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
25: (1.Ferry | $0 | $0 | $sched3.$0 | ($waitFQ1.FQ31 + $waitFQ2.FQ32 +
$'cooptFQ3.FQ))\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
26: (Ferry | $0 | $0 | $sched3.$0 | ($waitFQ1.FQ31 + $waitFQ2.FQ32 + $'cooptFQ3.FQ))
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
      = (($cooptFQ1.1.'sched1.1.Ferry + $cooptFQ2.1.'sched2.1.Ferry +
$cooptFQ3.1.'sched3.1.Ferry) | $0 | $0 | $sched3.$0 | ($waitFQ1.FQ31 + $waitFQ2.FQ32 +
$'cooptFQ3.FQ)) \{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
27: (1.'sched3.1.Ferry | $0 | $0 | $sched3.$0 | FQ)
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
28: ('sched3.1.Ferry | $0 | $0 | $sched3.$0 | ($waitFQ1.FQ1 + $waitFQ2.FQ2 + $waitFQ3.FQ3))
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
29: (1.Ferry | $0 | $0 | $0 | ($waitFQ1.FQ1 + $waitFQ2.FQ2 + $waitFQ3.FQ3))
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
30: (Ferry | $0 | $0 | $0 | ($waitFQ1.FQ1 + $waitFQ2.FQ2 +
$waitFQ3.FQ3))\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
      = (($cooptFQ1.1.'sched1.1.Ferry + $cooptFQ2.1.'sched2.1.Ferry +
$cooptFQ3.1.'sched3.1.Ferry) | $0 | $0 | $0 | ($waitFQ1.FQ1 + $waitFQ2.FQ2 + $waitFQ3.FQ3))
\{cooptFQ1,cooptFQ2,cooptFQ3,sched1,sched2,sched3,waitFQ1,waitFQ2,waitFQ3}
```

Figure 3.19

Model

```
bi Waiter $waitCQ.(valGet3.$0 + valGet0.Waiter + valGet1.Waiter\
+ valGet2.Waiter + valGet4.Waiter)

bi Val0 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4 + $'valGet0.Val0

bi Val1 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4 + $'valGet1.Val1

bi Val2 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4 + $'valGet2.Val2

bi Val3 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4 + $'valGet3.Val3

bi Val4 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4 + $'valGet4.Val4

bi Signaller valGet0.'valAss1.'waitCQ.Signaller +\
             valGet1.'valAss2.'waitCQ.Signaller +\
             valGet2.'valAss3.'waitCQ.Signaller +\
             valGet3.'valAss4.'waitCQ.Signaller +\
             valGet4.'valAss5.'waitCQ.Signaller

bi Model (Waiter | Signaller | Val0)\{waitCQ,\
valGet0,valGet1,valGet2,valGet3,valGet4,\
valAss0,valAss1,valAss2,valAss3,valAss4}
```

Output

```
Sim> random
For how many steps: 20
  --- t<valGet0> --->
  --- t<valAss1> --->
  --- t<waitCQ> --->
  --- t<valGet1> --->
  --- t<valGet1> --->
  --- t<valAss2> --->
  --- t<waitCQ> --->
  --- t<valGet2> --->
  --- t<valGet2> --->
  --- t<valAss3> --->
  --- t<waitCQ> --->
  --- t<valGet3> --->
  --- t<valGet3> --->
  --- t<valAss4> --->
  ** Simulation terminated: Deadlock. **

Simulated agent: ($0 | 'waitCQ.Signaller |
Val4)\{valAss0,valAss1,valAss2,valAss3,valAss4,valGet0,valGet1,valGet2,valGet3,valGet4,waitCQ}
Transitions:
  ** Deadlocked. **
```

Figure 3.20

Model

```
bi Waiter1 valGet3.0 + valGet0.'waitCQ1.Waiting1\
                     + valGet1.'waitCQ1.Waiting1\
                     + valGet2.'waitCQ1.Waiting1\
                     + valGet4.'waitCQ1.Waiting1

bi Waiting1 try1.(valGet3.'goGo1.0 + valGet0.'noGo1.Waiting1\
                                   + valGet1.'noGo1.Waiting1\
                                   + valGet2.'noGo1.Waiting1\
                                   + valGet4.'noGo1.Waiting1)

bi Waiter2 valGet3.0 + valGet0.'waitCQ2.Waiting2\
                     + valGet1.'waitCQ2.Waiting2\
                     + valGet2.'waitCQ2.Waiting2\
                     + valGet4.'waitCQ2.Waiting2
bi Waiting2 try2.(valGet3.'goGo2.0 + valGet0.'noGo2.Waiting2\
                                   + valGet1.'noGo2.Waiting2\
                                   + valGet2.'noGo2.Waiting2\
                                   + valGet4.'noGo2.Waiting2)
bi Val0 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4 + 'valGet0.Val0
bi Val1 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4 + 'valGet1.Val1
bi Val2 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4 + 'valGet2.Val2
bi Val3 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4 + 'valGet3.Val3
bi Val4 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4 + 'valGet4.Val4

bi Signaller valGet0.'valAss1.'signalCQ.done.Signaller +\
             valGet1.'valAss2.'signalCQ.done.Signaller +\
             valGet2.'valAss3.'signalCQ.done.Signaller +\
             valGet3.'valAss4.'signalCQ.done.Signaller +\
             valGet4.'valAss5.'signalCQ.done.Signaller

bi CQ signalCQ.CQ + waitCQ1.CQ1 + waitCQ2.CQ2
bi CQ1 signalCQ.Try1000 + waitCQ2.CQ12
bi CQ2 signalCQ.Try2000 + waitCQ1.CQ21
bi CQ12 signalCQ.Try1200
bi CQ21 signalCQ.Try2100
bi Try0000 'done.CQ
bi Try1000 'try1.(noGo1.'done.CQ1 + goGo1.Try0000)
bi Try0010 'done.CQ1
bi Try2000 'try2.(noGo2.'done.CQ2 + goGo2.Try0000)
bi Try0020 'done.CQ2
bi Try1200 'try1.(noGo1.'done.CQ12 + goGo1.Try2000)
bi Try0012 'done.CQ12
bi Try2100 'try2.(noGo2.'done.CQ21 + goGo2.Try1000)
bi Try0021 'done.CQ21
bi Try2010 'try2.(noGo2.'done.CQ12 + goGo2.Try0010)
bi Try1020 'try1.(noGo1.'done.CQ21 + goGo1.Try0020)
bi Model (Waiter1 | Waiter2 | Signaller | CQ | Val0)\{waitCQ1,waitCQ2,\
signalCQ,done,try1,try2,goGo1,goGo2,noGo1,noGo2,\
valGet0,valGet1,valGet2,valGet3,valGet4,\
valAss0,valAss1,valAss2,valAss3,valAss4,valAss5}
```

Output

```
Sim> random 30
  --- t<valGet0> --->
  --- t<valAss1> --->
  --- t<valGet1> --->
  --- t<waitCQ1> --->
  --- t<signalCQ> --->
  --- t<try1> --->
  --- t<valGet1> --->
  --- t<valGet1> --->
  --- t<noGo1> --->
  --- t<done> --->
  --- t<waitCQ2> --->
  --- t<valGet1> --->
  --- t<valAss2> --->
  --- t<signalCQ> --->
  --- t<try1> --->
  --- t<valGet2> --->
  --- t<noGo1> --->
  --- t<done> --->
  --- t<valGet2> --->
  --- t<valAss3> --->
  --- t<signalCQ> --->
  --- t<try1> --->
  --- t<valGet3> --->
  --- t<goGo1> --->
  --- t<try2> --->
  --- t<valGet3> --->
  --- t<goGo2> --->
  --- t<done> --->
  --- t<valGet3> --->
  --- t<valAss4> --->
Simulation complete.
```

Figure 3.21

Model

```
bi Waiter1 valGet3.0 + valGet0.'waitCQ1.Waiting1\
                     + valGet1.'waitCQ1.Waiting1\
                     + valGet2.'waitCQ1.Waiting1\
                     + valGet4.'waitCQ1.Waiting1

bi Waiting1 try1.(valGet3.'goGo1.0 + valGet0.'noGo1.Waiting1\
                                   + valGet1.'noGo1.Waiting1\
                                   + valGet2.'noGo1.Waiting1\
                                   + valGet4.'noGo1.Waiting1)

bi Waiter2 valGet3.0 + valGet0.'waitCQ2.Waiting2\
                     + valGet1.'waitCQ2.Waiting2\
                     + valGet2.'waitCQ2.Waiting2\
                     + valGet4.'waitCQ2.Waiting2

bi Waiting2 try2.(valGet3.'goGo2.0 + valGet0.'noGo2.Waiting2\
                                   + valGet1.'noGo2.Waiting2\
                                   + valGet2.'noGo2.Waiting2\
                                   + valGet4.'noGo2.Waiting2)

bi Val0 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4 + 'valGet0.Val0
bi Val1 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4 + 'valGet1.Val1
bi Val2 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4 + 'valGet2.Val2
bi Val3 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4 + 'valGet3.Val3
bi Val4 valAss1.Val1+valAss2.Val2+valAss3.Val3+valAss4.Val4 + 'valGet4.Val4

bi Signaller valGet0.'valAss1.'signalCQ.done.Signaller +\
             valGet1.'valAss2.'signalCQ.done.Signaller +\
             valGet2.'valAss3.'signalCQ.done.Signaller +\
             valGet3.'valAss4.'signalCQ.done.Signaller +\
             valGet4.'valAss5.'signalCQ.done.Signaller

bi CQ signalCQ.CQ + waitCQ1.CQ1 + waitCQ2.CQ2
bi CQ1 signalCQ.Try1000 + waitCQ2.CQ12
bi CQ2 signalCQ.Try2000 + waitCQ1.CQ21
bi CQ12 signalCQ.Try1200
bi CQ21 signalCQ.Try2100

bi Try0000 'done.CQ
bi Try1000 'try1.(noGo1.Try0010 + goGo1.Try0000)
bi Try0010 'done.CQ1
bi Try2000 'try2.(noGo2.Try0020 + goGo2.Try0000)
bi Try0020 'done.CQ2
bi Try1200 'try1.(noGo1.Try2010 + goGo1.Try2000)
bi Try0012 'done.CQ12
bi Try2100 'try2.(noGo2.Try1020 + goGo2.Try1000)
bi Try0021 'done.CQ21
bi Try2010 'try2.(noGo2.Try0012 + goGo2.Try0010)
bi Try1020 'try1.(noGo1.Try0021 + goGo1.Try0020)

bi Model (Waiter1 | Waiter2 | Signaller | CQ | Val0)\{waitCQ1,waitCQ2,\
signalCQ,done,try1,try2,goGo1,goGo2,noGo1,noGo2,\
valGet0,valGet1,valGet2,valGet3,valGet4,\
valAss0,valAss1,valAss2,valAss3,valAss4,valAss5}
```

Output

```
Sim> random 30
  --- t<valGet0> --->
  --- t<valGet0> --->
  --- t<valGet0> --->
  --- t<valAss1> --->
  --- t<waitCQ2> --->
  --- t<signalCQ> --->
  --- t<try2> --->
  --- t<valGet1> --->
  --- t<noGo2> --->
  --- t<done> --->
  --- t<waitCQ1> --->
  --- t<valGet1> --->
  --- t<valAss2> --->
  --- t<signalCQ> --->
  --- t<try2> --->
  --- t<valGet2> --->
  --- t<noGo2> --->
  --- t<try1> --->
  --- t<valGet2> --->
  --- t<noGo1> --->
  --- t<done> --->
  --- t<valGet2> --->
  --- t<valAss3> --->
  --- t<signalCQ> --->
  --- t<try2> --->
  --- t<valGet3> --->
  --- t<goGo2> --->
  --- t<try1> --->
  --- t<valGet3> --->
  --- t<goGo1> --->
Simulation complete.

Simulated agent: (0 | 0 | done.Signaller | Try0000 | Val3)
\{done,goGo1,goGo2,noGo1,noGo2,signalCQ,try1,try2,valAss0,valAss1,valAss2,valAss3,valAss4,valAss5
,valGet0,valGet1,valGet2,valGet3,valGet4,waitCQ1,waitCQ2}
Transitions:
    1: --- t<done> ---> (0 | 0 | Signaller | CQ | Val3)
\{done,goGo1,goGo2,noGo1,noGo2,signalCQ,try1,try2,valAss0,valAss1,valAss2,valAss3,valAss4,valAss5
,valGet0,valGet1,valGet2,valGet3,valGet4,waitCQ1,waitCQ2}
```

Figure 3.22

Model

```
bi Ited Checker10

bi Checker0 0
bi Checker1 1.(iGet3.Ited + iGet0.Checker0)
bi Checker2 1.(iGet3.Ited + iGet0.Checker1)
bi Checker3 1.(iGet3.Ited + iGet0.Checker2)
bi Checker4 1.(iGet3.Ited + iGet0.Checker3)
bi Checker5 1.(iGet3.Ited + iGet0.Checker4)
bi Checker6 1.(iGet3.Ited + iGet0.Checker5)
bi Checker7 1.(iGet3.Ited + iGet0.Checker6)
bi Checker8 1.(iGet3.Ited + iGet0.Checker7)
bi Checker9 1.(iGet3.Ited + iGet0.Checker8)
bi Checker10 1.(iGet3.Ited + iGet0.Checker9)

bi Iter 1.'iGet0.1.'iGet0.1.'1Get3.$0

bi Model (Ited | Iter)\{iGet3,iGet0}
```

Output

```
Command: if m322.cwb
done.
Command: states Model
1: Model
2: ((iGet3.Ited + iGet0.Checker9) | 'iGet0.1.'iGet0.1.'1Get3.$0)\{iGet0,iGet3}
3: (Checker9 | 1.'iGet0.1.'1Get3.$0)\{iGet0,iGet3}
4: ((iGet3.Ited + iGet0.Checker8) | 'iGet0.1.'1Get3.$0)\{iGet0,iGet3}
5: (Checker8 | 1.'1Get3.$0)\{iGet0,iGet3}
6: ((iGet3.Ited + iGet0.Checker7) | '1Get3.$0)\{iGet0,iGet3}
7: ((iGet3.Ited + iGet0.Checker7) | $0)\{iGet0,iGet3}

Command: statesobs Model
===  ===> Model
=== 1 ===> ((iGet3.Ited + iGet0.Checker9) | 'iGet0.1.'iGet0.1.'1Get3.$0)\{iGet0,iGet3}
=== 1 ===> (Checker9 | 1.'iGet0.1.'1Get3.$0)\{iGet0,iGet3}
=== 1 1 ===> ((iGet3.Ited + iGet0.Checker8) | 'iGet0.1.'1Get3.$0)\{iGet0,iGet3}
=== 1 1 ===> (Checker8 | 1.'1Get3.$0)\{iGet0,iGet3}
=== 1 1 1 ===> ((iGet3.Ited + iGet0.Checker7) | '1Get3.$0)\{iGet0,iGet3}
=== 1 1 1 '1Get3 ===> ((iGet3.Ited + iGet0.Checker7) | $0)\{iGet0,iGet3}
```

Figure 3.23

Model

See Figure 3.11

Output

See Figure 3.11.

Figure 3.24

Model

```
bi Philosopher 'rfAcq1.2.'lfAcq1.2.'rfRel1.'lfRel1.2.Philosopher
bi Fork fAcq1.NoFork
bi NoFork fRel1.Fork
bi P1 Philosopher[a1/rfAcq1,a2/lfAcq1,r1/rfRel1,r2/lfRel1]
bi P2 Philosopher[a2/rfAcq1,a3/lfAcq1,r2/rfRel1,r3/lfRel1]
bi P3 Philosopher[a3/rfAcq1,a1/lfAcq1,r3/rfRel1,r1/lfRel1]
bi Fork1 Fork[a1/fAcq1,r1/fRel1]
bi Fork2 Fork[a2/fAcq1,r2/fRel1]
bi Fork3 Fork[a3/fAcq1,r3/fRel1]
bi Model (Fork1 | Fork2 | Fork3 | P1 | P2 | P3)\{a1,a2,a3,r1,r2,r3}
```

Output

```
Command: sim
Agent: Model
Simulated agent: Model
Transitions:
    1: --- t<a3> ---> (Fork1 | Fork2 | NoFork[a3/fAcq1,r3/fRel1] | P1 | P2 |
(2.'lfAcq1.2.'rfRel1.'lfRel1.2.Philosopher)[a1/lfAcq1,r1/lfRel1,a3/rfAcq1,r3/rfRel1])\{a1,a2,a3,r
1,r2,r3}
    2: --- t<a2> ---> (Fork1 | NoFork[a2/fAcq1,r2/fRel1] | Fork3 | P1 |
(2.'lfAcq1.2.'rfRel1.'lfRel1.2.Philosopher)[a3/lfAcq1,r3/lfRel1,a2/rfAcq1,r2/rfRel1] |
P3)\{a1,a2,a3,r1,r2,r3}
    3: --- t<a1> ---> (NoFork[a1/fAcq1,r1/fRel1] | Fork2 | Fork3 |
(2.'lfAcq1.2.'rfRel1.'lfRel1.2.Philosopher)[a2/lfAcq1,r2/lfRel1,a1/rfAcq1,r1/rfRel1] | P2 |
P3)\{a1,a2,a3,r1,r2,r3}
Sim> random 12
  --- t<a3> --->
  --- t<a2> --->
  --- t<a1> --->
  ** Simulation terminated: Deadlock. **
Simulated agent: (NoFork[a1/fAcq1,r1/fRel1] | NoFork[a2/fAcq1,r2/fRel1] |
NoFork[a3/fAcq1,r3/fRel1] | (2.'lfAcq1.2.'rfRel1.'lfRel1.2.Philosopher)
[a2/lfAcq1,r2/lfRel1,a1/rfAcq1,r1/rfRel1] |
(2.'lfAcq1.2.'rfRel1.'lfRel1.2.Philosopher)[a3/lfAcq1,r3/lfRel1,a2/r
```

Figure 3.26/7/8

Model

```
bi Boat $'tugacq2.$'jacq1.(Work|NewBoat)
bi Work 3.'tugrel2.10.$'tugacq1.3.'tugrel1.'jrel1.Idle
bi NewBoat 4.'n.Boat
bi Idle 1.Idle
bi Tugs3 ($tugacq1.Tugs2)+($tugacq2.Tugs1)+($tugacq3.Tugs0)
bi Tugs2 ($tugacq1.Tugs1)+($tugacq2.Tugs0)+($tugrel1.Tugs3)
bi Tugs1 ($tugacq1.Tugs0)+($tugrel1.Tugs2)+($tugrel2.Tugs3)
bi Tugs0 ($tugrel1.Tugs1)+($tugrel2.Tugs2)+($tugrel3.Tugs3)
bi Jetty2 ($jacq1.Jetty1) + ($jacq2.Jetty0)
bi Jetty1 ($jacq1.Jetty0) + ($jrel1.Jetty2)
bi Jetty0 ($jrel1.Jetty1) + ($jrel2.Jetty2)
bi Obs $n.Obs
bi DEMOS Obs|100.0
bi Model (Tugs3|Jetty2|Boat)
     \{tugacq1,tugacq2,tugacq3,tugrel1,tugrel2,tugrel3,jacq1,jacq2,jrel1,jrel2}
bi Prog (DEMOS | Model)\{n}
```

Output

```
Sim>   --- t<tugacq2> --->
  --- 1 --->
  --- t<jacq1> --->
  --- 1 --->
  --- 1 --->
  --- 1 --->
  --- t<tugrel2> --->
  --- 1 --->
  --- t<n> --->
  --- t<tugacq2> --->
  --- t<jacq1> --->
  --- 1 --->
  --- 1 --->
  --- 1 --->
  --- t<tugrel2> --->
  --- 1 --->
  --- t<n> --->
  --- t<tugacq2> --->
  --- 1 --->
  --- 1 --->
  --- 1 --->
  --- 1 --->
  --- 1 --->
  --- t<tugacq1> --->
```

Chapter 6

Figure 6.1

Model

```
bi Boat 'jA1.'tA2.'tR2.'tA1.'jR1.0
bi T2 (tA1.T1) + (tA2.T0)
bi T1 (tA1.T0) + (tR1.T2)
bi T0 (tR1.T1) + (tR2.T2)
bi J2 (jA1.J1) + (jA2.J0)
bi J1 (jA1.J0) + (jR1.J2)
bi J0 (jR1.J1) + (jR2.J2)
bi M1 (T2|J2|Boat|Boat|Boat)\{tA1,tA2,jA1,jA2,tR1,tR2,jR1,jR2}
bi Js2 (jA1.Js1)
bi Js1 (jA1.Js0) + (jR1.Js2)
bi Js0 (jR1.Js1)
bi M2 (T2|Js2|Boat|Boat|Boat)\{tA1,tA2,tR1,tR2,jA1,jR1}
```

Output

```
Command: eq
Agent: M1
Agent: M2
true
Command: cong
Agent: M1
Agent: M2
true
```

Figure 6.3

Model

```
bi R 'bA1.'bR1.R
bi W 'bA3.'bR3.W
bi B3 (bA1.B2) +(bA2.B1) + (bA3.B0)
bi B2 (bA1.B1) +(bA2.B0) + (bR1.B3)
bi B1 (bA1.B0) +(bR2.B3) + (bR1.B2)
bi B0 (bR3.B3) +(bR2.B2) + (bR1.B1)
bi M1 (R|R|W|B3)\{bR1,bR2,bR3,bA1,bA2,bA3}
bi ER 'bRA1.'bRR1.ER
bi EW 'bWA3.'bWR3.EW
bi EM (B3|ER[bA1/bRA1,bR1/bRR1]|ER[bA1/bRA1,bR1/bRR1]|EW[bA3/bWA3,bR3/bWR3])\
\{bA1,bA2,bA3,bR1,bR2,bR3}
bi SB3 (bSA1.SB2) + (bSA3.SB0)
bi SB2 (bSA1.SB1) + (bSR1.SB3)
bi SB1 (bSA1.SB0) + (bSR1.SB2)
bi SB0 (bSR3.SB3) + (bSR1.SB1)
bi EM1 (SB3|ER[bSA1/bRA1,bSR1/bRR1]|ER[bSA1/bRA1,bSR1/bRR1]|\
EW[bSA3/bWA3,bSR3/bWR3])\
\{bSA1,bSA3,bSR1,bSR3}
```

Output

```
Command: eq
Agent: EM
Agent: EM1
true
Command: cong
Agent: EM
Agent: EM1
true
Command: eq
Agent: M1
Agent: EM1
true
Command: cong
Agent: M1
Agent: EM1
true
```

Chapter 6

Figure 6.1

Model

```
bi Boat 'jA1.'tA2.'tR2.'tA1.'jR1.0
bi T2 (tA1.T1) + (tA2.T0)
bi T1 (tA1.T0) + (tR1.T2)
bi T0 (tR1.T1) + (tR2.T2)
bi J2 (jA1.J1) + (jA2.J0)
bi J1 (jA1.J0) + (jR1.J2)
bi J0 (jR1.J1) + (jR2.J2)
bi M1 (T2|J2|Boat|Boat|Boat)\{tA1,tA2,jA1,jA2,tR1,tR2,jR1,jR2}
bi Js2 (jA1.Js1)
bi Js1 (jA1.Js0) + (jR1.Js2)
bi Js0 (jR1.Js1)
bi M2 (T2|Js2|Boat|Boat|Boat)\{tA1,tA2,tR1,tR2,jA1,jR1}
```

Output

```
Command: eq
Agent: M1
Agent: M2
true
Command: cong
Agent: M1
Agent: M2
true
```

Figure 6.3

Model

```
bi R 'bA1.'bR1.R
bi W 'bA3.'bR3.W
bi B3 (bA1.B2) +(bA2.B1) + (bA3.B0)
bi B2 (bA1.B1) +(bA2.B0) + (bR1.B3)
bi B1 (bA1.B0) +(bR2.B3) + (bR1.B2)
bi B0 (bR3.B3) +(bR2.B2) + (bR1.B1)
bi M1 (R|R|W|B3)\{bR1,bR2,bR3,bA1,bA2,bA3}
bi ER 'bRA1.'bRR1.ER
bi EW 'bWA3.'bWR3.EW
bi EM (B3|ER[bA1/bRA1,bR1/bRR1]|ER[bA1/bRA1,bR1/bRR1]|EW[bA3/bWA3,bR3/bWR3])\
\{bA1,bA2,bA3,bR1,bR2,bR3}
bi SB3 (bSA1.SB2) + (bSA3.SB0)
bi SB2 (bSA1.SB1) + (bSR1.SB3)
bi SB1 (bSA1.SB0) + (bSR1.SB2)
bi SB0 (bSR3.SB3) + (bSR1.SB1)
bi EM1 (SB3|ER[bSA1/bRA1,bSR1/bRR1]|ER[bSA1/bRA1,bSR1/bRR1]|\
EW[bSA3/bWA3,bSR3/bWR3])\
\{bSA1,bSA3,bSR1,bSR3}
```

Output

```
Command: eq
Agent: EM
Agent: EM1
true
Command: cong
Agent: EM
Agent: EM1
true
Command: eq
Agent: M1
Agent: EM1
true
Command: cong
Agent: M1
Agent: EM1
true
```

Figure 6.5

Model

```
bi M 'dA2.'dR2.hD.M
bi P 'dA4.'dR2.'hD.'dR2.P
bi D5 dA5.D0 + dA4.D1 + dA3.D2 + dA2.D3 + dA1.D4
bi D4 dA4.D0 + dA3.D1 + dA2.D2 + dA1.D3 + dR1.D5
bi D3 dA3.D0 + dA2.D1 + dA1.D2 + dR2.D5 + dR1.D4
bi D2 dA2.D0 + dA1.D1 + dR3.D5 + dR2.D4 + dR1.D3
bi D1 dA1.D0 + dR4.D5 + dR3.D4 + dR2.D3 + dR1.D2
bi D0 dR5.D5 + dR4.D4 + dR3.D3 + dR2.D2 + dR1.D1
bi F (M|P|D5)\{dR1,dR2,dR3,dR4,dR5,dA1,dA2,dA3,dA4,dA5,hD}

bi D15 d1A4.D11+ d1A2.D13
bi D13 d1R2.D15 + d1A2.D11
bi D11 d1R2.D13
bi F1 (M[d1A2/dA2,d1R2/dR2]|P[d1A4/dA4,d1R2/dR2]|D15)\{d1R2,d1A2,d1A4,hD}
```

Output

```
Agent: F
Agent: F1
true
Command: eq
Agent: F
Agent: F1
true
```

Figure 6.8

Model

```
bi Bureau 'typing.C1 + 'copying.C2 + 'printing.C3
bi Messenger typing.D1 + copying.D2

bi C1 0
bi C2 0
bi C3 0
bi D1 0
bi D2 0

bi Model (Messenger | Bureau)\{typing,copying,printing}

bi Model1 (Messenger | Bureau)
```

Output

```
Command: states Model1
1: D1 | C3
2: D1 | C2
3: D2 | C1
4: D2 | C3
5: Messenger | C1
6: Messenger | C3
7: Messenger | C2
8: D1 | Bureau
9: D2 | Bureau
10: D2 | C2
11: D1 | C1
12: Model1

Command: states Model
1: (D2 | C2)\{copying,printing,typing}
2: (D1 | C1)\{copying,printing,typing}
3: Model
```

Figure 6.9

Model

```
bi Bureau 'typing.C1 + 'copying.C2 + 'printing.C3
bi Messenger typing.D1 + copying.D2

bi C1 0
bi C2 0
bi C3 0
bi D1 0
bi D2 0

bi Model (Messenger | Bureau)\{typing,copying,printing}

bi Model1 (Messenger | Bureau)

bi Emergency 'typing.Bureau

bi Problem (Messenger | Emergency)\{typing,copying,printing}

bi Problem1 (Messenger | Emergency)
```

Output

```
Command: states Problem
1: (D1 | Bureau)\{copying,printing,typing}
2: Problem

Command: states Problem1
1: D2 | C1
2: D2 | C3
3: Messenger | C1
4: Messenger | C3
5: Messenger | C2
6: D2 | C2
7: D2 | Bureau
8: D1 | C1
9: D1 | C3
10: D1 | C2
11: Messenger | Bureau
12: D1 | Emergency
13: D2 | Emergency
14: D1 | Bureau
15: Problem1
```

Figure 6.11

First simplification - Model

```
bi Stream 'memAcq4.'memRel4.'tSched.Stream

bi Mem4 memAcq4.Mem0
bi Mem0 memRel4.Mem4

bi Input (Stream | Mem4)\{memAcq4,memRel4}

bi Modela (Input | Input )

bi Stream1 'tSched.Stream1

bi Input1 Stream1

bi Model1a (Input1|Input1)
```

Output

```
Command: eq
Agent: Modela
Agent: Model1a
true
```

Second simplification - Model

```
bi Trans tsched.'linkAcq1.'buffAcq2.'linkRel1.\
'linkAcq1.'buffRel2.'linkRel1.Trans

bi Link1 linkAcq1.Link0
bi Link0 linkRel1.Link1

bi Buffs2 buffAcq2.Buffs0
bi Buffs0 buffRel2.Buffs2

bi Modelb (Trans | Trans | Link1 | Buffs2)\
\{linkAcq1,linkRel1,buffAcq2,buffRel2}


bi Trans1 tsched.'linkAcq1.'buffAcq2.'buffRel2.'linkRel1.Trans

bi Model1b (Trans1 | Trans1 | Link1 | Buffs2)\
\{linkAcq1,linkRel1,buffAcq2,buffRel2}
```

Output

```
Command: eq
Agent: Modelb
Agent: Model1b
false
```

Figure 6.12

Model

```
bi Station 'eAcq1.Sending
bi Sending 'eRel1.Station

bi Ether1 eAcq1.Ether0
bi Ether0 eRel1.Ether1

bi Model (Ether1 | Station | Station | Station)\{eAcq1,eRel1}
```

Output

```
Command: states Model
1: Model
      = (Ether1 | Station | Station | Station)\{eAcq1,eRel1}
2: (Ether0 | Sending | Station | Station)\{eAcq1,eRel1}
      = (Ether0 | Station | Sending | Station)\{eAcq1,eRel1}
      = (Ether0 | Station | Station | Sending)\{eAcq1,eRel1}
```

Figure 6.14

Model

```
bi Station1 'eWaitUntil1.Waiting1
bi Waiting1 $sched1.Trying1
bi Trying1 eLen1.1.Station1 + eLen2.1.Station1 +eLen3.1.Station1\
        + eLen0.'eAcq1.Sending1
bi Sending1 4.'eRel1.Done1
bi Done1 'eQSignal.Station1

bi Station2 'eWaitUntil2.Waiting2
bi Waiting2 $sched2.Trying2
bi Trying2 eLen1.1.Station2 + eLen2.1.Station2 +eLen3.1.Station2\
        + eLen0.'eAcq1.Sending2
bi Sending2 4.'eRel1.Done2
bi Done2 'eQSignal.Station2

bi Station3 'eWaitUntil3.Waiting3
bi Waiting3 $sched3.Trying3
bi Trying3 eLen1.1.Station3 + eLen2.1.Station3 +eLen3.1.Station3\
        + eLen0.'eAcq1.Sending3
bi Sending3 4.'eRel1.Done3
bi Done3 'eQSignal.Station3

bi Ether1 $eAcq1.Ether0
bi Ether0 $eRel1.Ether1

bi EtherQ0000 $eWaitUntil1.EtherQ1001 + $eWaitUntil2.EtherQ2001\
          + $eWaitUntil3.EtherQ3001 + $eQSignal.Signal0000

bi EtherQ1001 $eWaitUntil2.EtherQ1202 + $eWaitUntil3.EtherQ1302\
                                      + $eQSignal.Signal1001
bi EtherQ2001 $eWaitUntil1.EtherQ2102 + $eWaitUntil3.EtherQ2302\
                                    + $eQSignal.Signal2001
bi EtherQ3001 $eWaitUntil2.EtherQ3202 + $eWaitUntil1.EtherQ3102\
                                    + $eQSignal.Signal3001

bi EtherQ1202 $eWaitUntil3.EtherQ1233 + $eQSignal.Signal1202
bi EtherQ1302 $eWaitUntil2.EtherQ1323 + $eQSignal.Signal1302
bi EtherQ2102 $eWaitUntil3.EtherQ2133 + $eQSignal.Signal2102
bi EtherQ2302 $eWaitUntil1.EtherQ2313 + $eQSignal.Signal2302
bi EtherQ3102 $eWaitUntil2.EtherQ3123 + $eQSignal.Signal3102
bi EtherQ3202 $eWaitUntil1.EtherQ3213 + $eQSignal.Signal3202

bi EtherQ1233 $eQSignal.Signal1233
bi EtherQ1323 $eQSignal.Signal1323
bi EtherQ2133 $eQSignal.Signal2133
bi EtherQ2313 $eQSignal.Signal2313
bi EtherQ3123 $eQSignal.Signal3123
bi EtherQ3213 $eQSignal.Signal3213

bi Signal0000 EtherQ0000

bi Signal1001 'sched1.'eLen0.Signal0000
bi Signal2001 'sched2.'eLen0.Signal0000
bi Signal3001 'sched3.'eLen0.Signal0000

bi Signal1202 'sched1.'eLen1.Signal2001
bi Signal1302 'sched1.'eLen1.Signal3001
bi Signal2102 'sched2.'eLen1.Signal1001
bi Signal2302 'sched2.'eLen1.Signal3001
bi Signal3102 'sched3.'eLen1.Signal1001
bi Signal3202 'sched3.'eLen1.Signal2001

bi Signal1233 'sched1.'eLen2.Signal2302
bi Signal1323 'sched1.'eLen2.Signal3202
bi Signal2133 'sched2.'eLen2.Signal1302
bi Signal2313 'sched2.'eLen2.Signal3102
bi Signal3123 'sched3.'eLen2.Signal1202
bi Signal3213 'sched3.'eLen2.Signal2102

bi Model (Ether0 | EtherQ1202 | Waiting1 | Waiting2 | Sending3)\
\{eAcq1,eRel1,eLen0,eLen1,eLen2,eLen3,sched1,sched2,sched3,eQSignal,\
eWaitUntil1,eWaitUntil2,eWaitUntil3}
```

Output

```
Command: states Model
1: Model
2: ($eRel1.Ether1 | ($eQSignal.Signal1202 + $eWaitUntil3.EtherQ1233) | $sched1.Trying1 |
$sched2.Trying2 |
3.'eRel1.Done3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3
,sched1,sched2,sched3}
```

```
      = ($eRel1.Ether1 | EtherQ1202 | $sched1.Trying1 | Waiting2 |
3.'eRel1.Done3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3
,sched1,sched2,sched3}
3: ($eRel1.Ether1 | ($eQSignal.Signal1202 + $eWaitUntil3.EtherQ1233) | $sched1.Trying1 |
$sched2.Trying2 |
2.'eRel1.Done3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3
,sched1,sched2,sched3}
4: ($eRel1.Ether1 | ($eQSignal.Signal1202 + $eWaitUntil3.EtherQ1233) | $sched1.Trying1 |
$sched2.Trying2 |
1.'eRel1.Done3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3
,sched1,sched2,sched3}
5: ($eRel1.Ether1 | ($eQSignal.Signal1202 + $eWaitUntil3.EtherQ1233) | $sched1.Trying1 |
$sched2.Trying2 |
'eRel1.Done3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,s
ched1,sched2,sched3}
6: (Ether1 | ($eQSignal.Signal1202 + $eWaitUntil3.EtherQ1233) | $sched1.Trying1 | $sched2.Trying2
|
Done3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sched1,s
ched2,sched3}
7: (Ether1 | Signal1202 | $sched1.Trying1 | $sched2.Trying2 |
Station3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sched
1,sched2,sched3}
8: (Ether1 | 'eLen1.Signal2001 | Trying1 | $sched2.Trying2 |
Station3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sched
1,sched2,sched3}
9: (Ether1 | Signal2001 | 1.Station1 | $sched2.Trying2 |
Station3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sched
1,sched2,sched3}
10: (Ether1 | 'eLen0.Signal0000 | 1.Station1 | Trying2 |
Station3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sched
1,sched2,sched3}
11: (Ether1 | Signal0000 | 1.Station1 | 'eAcq1.Sending2 |
Station3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sched
1,sched2,sched3}
12: (Ether0 | Signal0000 | 1.Station1 | Sending2 |
Station3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sched
1,sched2,sched3}
      = (Ether1 | EtherQ3001 | 1.Station1 | 'eAcq1.Sending2 |
Waiting3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sched
1,sched2,sched3}
13: (Ether0 | EtherQ3001 | 1.Station1 | Sending2 |
Waiting3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sched
1,sched2,sched3}
14: ($eRel1.Ether1 | ($eQSignal.Signal3001 + $eWaitUntil1.EtherQ3102 + $eWaitUntil2.EtherQ3202) |
Station1 | 3.'eRel1.Done2 |
$sched3.Trying3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil
3,sched1,sched2,sched3}
15: ($eRel1.Ether1 | EtherQ3102 | Waiting1 | 3.'eRel1.Done2 |
$sched3.Trying3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil
3,sched1,sched2,sched3}
16: ($eRel1.Ether1 | ($eQSignal.Signal3102 + $eWaitUntil2.EtherQ3123) | $sched1.Trying1 |
2.'eRel1.Done2 |
$sched3.Trying3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil
3,sched1,sched2,sched3}
17: ($eRel1.Ether1 | ($eQSignal.Signal3102 + $eWaitUntil2.EtherQ3123) | $sched1.Trying1 |
1.'eRel1.Done2 |
$sched3.Trying3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil
3,sched1,sched2,sched3}
18: ($eRel1.Ether1 | ($eQSignal.Signal3102 + $eWaitUntil2.EtherQ3123) | $sched1.Trying1 |
'eRel1.Done2 |
$sched3.Trying3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil
3,sched1,sched2,sched3}
19: (Ether1 | ($eQSignal.Signal3102 + $eWaitUntil2.EtherQ3123) | $sched1.Trying1 | Done2 |
$sched3.Trying3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil
3,sched1,sched2,sched3}
20: (Ether1 | Signal3102 | $sched1.Trying1 | Station2 |
$sched3.Trying3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil
3,sched1,sched2,sched3}
21: (Ether1 | 'eLen1.Signal1001 | $sched1.Trying1 | Station2 |
Trying3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sched1
,sched2,sched3}
22: (Ether1 | Signal1001 | $sched1.Trying1 | Station2 |
1.Station3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sch
ed1,sched2,sched3}
23: (Ether1 | 'eLen0.Signal0000 | Trying1 | Station2 |
1.Station3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sch
ed1,sched2,sched3}
24: (Ether1 | Signal0000 | 'eAcq1.Sending1 | Station2 |
1.Station3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sch
ed1,sched2,sched3}
25: (Ether0 | Signal0000 | Sending1 | Station2 |
1.Station3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sch
ed1,sched2,sched3}
      = (Ether1 | EtherQ2001 | 'eAcq1.Sending1 | Waiting2 |
1.Station3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sch
ed1,sched2,sched3}
26: (Ether0 | EtherQ2001 | Sending1 | Waiting2 |
1.Station3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sch
ed1,sched2,sched3}
27: ($eRel1.Ether1 | ($eQSignal.Signal2001 + $eWaitUntil1.EtherQ2102 + $eWaitUntil3.EtherQ2302) |
3.'eRel1.Done1 | $sched2.Trying2 |
```

```
Station3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sched
1,sched2,sched3}
28: ($eRel1.Ether1 | EtherQ2302 | 3.'eRel1.Done1 | $sched2.Trying2 |
Waiting3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sched
1,sched2,sched3}
29: ($eRel1.Ether1 | ($eQSignal.Signal2302 + $eWaitUntil1.EtherQ2313) | 2.'eRel1.Done1 |
$sched2.Trying2 |
$sched3.Trying3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil
3,sched1,sched2,sched3}
30: ($eRel1.Ether1 | ($eQSignal.Signal2302 + $eWaitUntil1.EtherQ2313) | 1.'eRel1.Done1 |
$sched2.Trying2 |
$sched3.Trying3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil
3,sched1,sched2,sched3}
31: ($eRel1.Ether1 | ($eQSignal.Signal2302 + $eWaitUntil1.EtherQ2313) | 'eRel1.Done1 |
$sched2.Trying2 |
$sched3.Trying3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil
3,sched1,sched2,sched3}
32: (Ether1 | ($eQSignal.Signal2302 + $eWaitUntil1.EtherQ2313) | Done1 | $sched2.Trying2 |
$sched3.Trying3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil
3,sched1,sched2,sched3}
33: (Ether1 | Signal2302 | Station1 | $sched2.Trying2 |
$sched3.Trying3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil
3,sched1,sched2,sched3}
34: (Ether1 | 'eLen1.Signal3001 | Station1 | Trying2 |
$sched3.Trying3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil
3,sched1,sched2,sched3}
35: (Ether1 | Signal3001 | Station1 | 1.Station2 |
$sched3.Trying3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil
3,sched1,sched2,sched3}
36: (Ether1 | 'eLen0.Signal0000 | Station1 | 1.Station2 |
Trying3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sched1
,sched2,sched3}
37: (Ether1 | Signal0000 | Station1 | 1.Station2 |
'eAcq1.Sending3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil
3,sched1,sched2,sched3}
38: (Ether0 | Signal0000 | Station1 | 1.Station2 |
Sending3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sched
1,sched2,sched3}
      = (Ether1 | EtherQ1001 | Waiting1 | 1.Station2 |
'eAcq1.Sending3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil
3,sched1,sched2,sched3}
39: (Ether0 | EtherQ1001 | Waiting1 | 1.Station2 |
Sending3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3,sched
1,sched2,sched3}
40: ($eRel1.Ether1 | ($eQSignal.Signal1001 + $eWaitUntil2.EtherQ1202 + $eWaitUntil3.EtherQ1302) |
$sched1.Trying1 | Station2 |
3.'eRel1.Done3)\{eAcq1,eLen0,eLen1,eLen2,eLen3,eQSignal,eRel1,eWaitUntil1,eWaitUntil2,eWaitUntil3
,sched1,sched2,sched3}
```

Figure 6.16

Model

```
bi Station1 'sAcq1.Trying1
bi Trying1 sAvail1.Station1 + sAvail0.'eQWait1.Waiting1
bi Waiting1 $eSched1.(cAvail0.'etAcq1.4.Done1\
 + cAvail1.BackOff1 + cAvail2.BackOff1 + cAvail3.BackOff1)
bi Done1 'sRel1.'etRel1.Trying1
bi BackOff1 'cRem1.$cAvail0.1.Trying1

bi Station2 'sAcq1.Trying2
bi Trying2 sAvail1.Station2 + sAvail0.'eQWait2.Waiting2
bi Waiting2 $eSched2.(cAvail0.'etAcq1.4.Done2\
 + cAvail1.BackOff2 + cAvail2.BackOff2 + cAvail3.BackOff2)
bi Done2 'sRel1.'etRel1.Trying2
bi BackOff2 'cRem1.$cAvail0.1.Trying2

bi Station3 'sAcq1.Trying3
bi Trying3 sAvail1.Station3 + sAvail0.'eQWait3.Waiting3
bi Waiting3 eSched3.(cAvail0.'etAcq1.4.Done3\
 + cAvail1.BackOff3 + cAvail2.BackOff3 + cAvail3.BackOff3)
bi Done3 'sRel1.'etRel1.Trying3
bi BackOff3 'cRem1.$cAvail0.1.Trying3

bi Ethernet $eQCoopt1.$'etAcq1.Used1 + $eQCoopt2.$'etAcq1.Used2 +\
           $eQCoopt3.$'etAcq1.Used3
bi Used1 eQLen0.Next1 + eQLen1.'cAdd1.Next1 + eQLen2.'cAdd1.Next1
bi Next1 'etRel1.'eSched1.ReSched
bi Used2 eQLen0.Next2 + eQLen1.'cAdd1.Next2 + eQLen2.'cAdd1.Next2
bi Next2 'etRel1.'eSched2.ReSched
bi Used3 eQLen0.Next3 + eQLen1.'cAdd1.Next3 + eQLen2.'cAdd1.Next3
bi Next3 'etRel1.'eSched3.ReSched
bi ReSched eQLen0.Ethernet + eQLen1.'cAdd1.(eQCoopt1.'eSched1.ReSched\
 + eQCoopt2.'eSched2.ReSched + eQCoopt3.'eSched3.ReSched)\
 + eQLen2.'cAdd1.(eQCoopt1.'eSched1.ReSched\
 + eQCoopt2.'eSched2.ReSched + eQCoopt3.'eSched3.ReSched)

bi Sending1 $sAcq1.Sending0 + $'sAvail1.Sending1
bi Sending0 $sRel1.Sending0 + $'sAvail0.Sending0

bi Cols0 $cAdd1.Cols1 + $cAdd2.Cols2 + $cAdd3.Cols3 + $'cAvail0.Cols0
bi Cols1 $cAdd1.Cols2 + $cAdd2.Cols3 + $cRem1.Cols0 + $'cAvail1.Cols1
bi Cols2 $cAdd1.Cols3 + $cRem1.Cols1 + $'cAvail2.Cols2
bi Cols3 $cRem1.Cols2+ $'cAvail3.Cols3


bi EtherQ0000 $eQWait1.EtherQ1001 + $eQWait2.EtherQ2001\
            + $eQWait3.EtherQ3001 + $'eQLen0.EtherQ0000

bi EtherQ1001 $eQWait2.EtherQ1202 + $eQWait3.EtherQ1302\
                                  + $'eQCoopt1.EtherQ0000 + $'eQLen1.EtherQ1001
bi EtherQ2001 $eQWait1.EtherQ2102 + $eQWait3.EtherQ2302\
                                  + $'eQCoopt2.EtherQ0000 + $'eQLen1.EtherQ2001
bi EtherQ3001 $eQWait2.EtherQ3202 + $eQWait1.EtherQ3102\
                                  + $'eQCoopt3.EtherQ0000 + $'eQLen1.EtherQ3001

bi EtherQ1202 $eQWait3.EtherQ1233 + $'eQCoopt1.EtherQ2001 + $'eQLen2.EtherQ1202
bi EtherQ1302 $eQWait2.EtherQ1323 + $'eQCoopt1.EtherQ3001 + $'eQLen2.EtherQ1302
bi EtherQ2102 $eQWait3.EtherQ2133 + $'eQCoopt2.EtherQ1001 + $'eQLen2.EtherQ2102
bi EtherQ2302 $eQWait1.EtherQ2313 + $'eQCoopt2.EtherQ3001 + $'eQLen2.EtherQ2302
bi EtherQ3102 $eQWait2.EtherQ3123 + $'eQCoopt3.EtherQ1001 + $'eQLen2.EtherQ3102
bi EtherQ3202 $eQWait1.EtherQ3213 + $'eQCoopt3.EtherQ2001 + $'eQLen2.EtherQ3202

bi EtherQ1233 $'eQCoopt1.EtherQ2302 + $'eQLen3.EtherQ1233
bi EtherQ1323 $'eQCoopt1.EtherQ3202 + $'eQLen3.EtherQ1323
bi EtherQ2133 $'eQCoopt2.EtherQ1302 + $'eQLen3.EtherQ2133
bi EtherQ2313 $'eQCoopt2.EtherQ3102 + $'eQLen3.EtherQ2313
bi EtherQ3123 $'eQCoopt3.EtherQ1202 + $'eQLen3.EtherQ3123
bi EtherQ3213 $'eQCoopt3.EtherQ2102 + $'eQLen3.EtherQ3213

bi EtherR1 $etAcq1.EtherR0
bi EtherR0 $etRel1.EtherR1

bi Transmitter1 (Station1 | Sending1)\{sAcq1,sRel1,sAvail0,sAvail1}
bi Transmitter2 (Station2 | Sending1)\{sAcq1,sRel1,sAvail0,sAvail1}
bi Transmitter3 (Station3 | Sending1)\{sAcq1,sRel1,sAvail0,sAvail1}

bi Model (EtherR0 | Ethernet | EtherQ1202 |\
        Cols0 | Waiting1 | Waiting2 | (Done3|Sending0)\{sAcq1,sRel1,sAvail0,sAvail1})\
\{etAcq1,etRel1,eLen0,eLen1,eLen2,eLen3,eSched1,eSched2,eSched3,\
eQCoopt1,eQCoopt2,eQCoopt3,eQWait1,eQWait2,eQWait3,eQLen3,eQLen2,eQLen1,eQLen0, \
cAdd1,cAdd2,cAdd3,cRem1,cAvail0,cAvail1,cAvail2,cAvail3}
```

Output

```
Sim> random 40
  --- t<sRel1> --->
  --- t<etRel1> --->
  --- t<eQCoopt1> --->
  --- t<sAvail0> --->
  --- t<eQWait3> --->
  --- t<etAcq1> --->
  --- t<eQLen2> --->
  --- t<cAdd1> --->
  --- t<etRel1> --->
  --- t<eSched1> --->
  --- t<cAvail1> --->
  --- t<eQLen2> --->
  --- t<cAdd1> --->
  --- t<cRem1> --->
  --- t<eQCoopt2> --->
  --- t<eSched2> --->
  --- t<eQLen1> --->
  --- t<cAdd1> --->
  --- t<eQCoopt3> --->
  --- t<eSched3> --->
  --- t<eQLen0> --->
  --- t<cAvail2> --->
  --- t<cAvail2> --->
  --- t<cRem1> --->
  --- t<cRem1> --->
  --- 1 --->
  --- 1 --->
  --- t<cAvail0> --->
  --- t<cAvail0> --->
  --- t<cAvail0> --->
  --- 1 --->
  --- t<sAvail0> --->
  --- sAvail1 --->
  --- 'sAcq1 --->
  --- t<eQWait3> --->
  --- sAvail1 --->
  --- 'sAcq1 --->
  --- sAvail0 --->
  --- t<eQCoopt3> --->
  --- t<eQWait1> --->
Simulation complete.


Command: states Ethernet
1: eQCoopt1.'eSched1.ReSched + eQCoopt2.'eSched2.ReSched + eQCoopt3.'eSched3.ReSched
2: 'cAdd1.(eQCoopt1.'eSched1.ReSched + eQCoopt2.'eSched2.ReSched + eQCoopt3.'eSched3.ReSched)
3: ReSched
4: 'eSched3.ReSched
5: 'eSched2.ReSched
6: 'eSched1.ReSched
7: 'cAdd1.Next3
8: Next3
9: 'cAdd1.Next2
10: Next2
11: 'cAdd1.Next1
12: Next1
13: Used3
14: Used2
15: Used1
16: $eQCoopt1.$'etAcq1.Used1 + $eQCoopt2.$'etAcq1.Used2 + $eQCoopt3.$'etAcq1.Used3
17: $'etAcq1.Used3
18: $'etAcq1.Used2
19: $'etAcq1.Used1
20: Ethernet


Command: states Station1
1: 'etRel1.Trying1
2: Done1
3: 1.Done1
4: 2.Done1
5: 1.Trying1
6: 3.Done1
7: $cAvail0.1.Trying1
8: 4.Done1
9: BackOff1
10: 'etAcq1.4.Done1
11: $eSched1.(cAvail0.'etAcq1.4.Done1 + cAvail1.BackOff1 + cAvail2.BackOff1 + cAvail3.BackOff1)
12: cAvail0.'etAcq1.4.Done1 + cAvail1.BackOff1 + cAvail2.BackOff1 + cAvail3.BackOff1
13: Waiting1
14: 'eQWait1.Waiting1
15: Trying1
16: Station1
```

Figure 6.18

Model

```
bi Reader 'bufacq1.'bufrel1.Reader
bi Writer 'bufacq3.'bufrel3.Writer
bi Buffers3 bufacq1.Buffers2 + bufacq3.Buffers0
bi Buffers2 bufacq1.Buffers1 + bufrel1.Buffers3
bi Buffers1 bufacq1.Buffers0 + bufrel1.Buffers2
bi Buffers0 bufrel1.Buffers1 + bufrel3.Buffers3
bi Model (Buffers3|Reader|Reader|Writer)\{bufacq1,bufacq3,bufrel1,bufrel3}
```

Output

```
Command: states Model
1: Model
2: (Buffers2 | Thinker0 | 2.Reader0 | 1.Writer)\{bufacq1,bufacq3,bufrel1,bufrel3}
3: (($bufacq1.Buffers1 + $bufrel1.Buffers3) | 2.Thinker1 | 1.Reader0 |
Writer)\{bufacq1,bufacq3,bufrel1,bufrel3}
      = (Buffers2 | Reader1 | 2.Thinker1 |
$'bufacq3.'bufrel3.Writer)\{bufacq1,bufacq3,bufrel1,bufrel3}
4: (($bufacq1.Buffers1 + $bufrel1.Buffers3) | 1.Thinker1 | Reader0 |
$'bufacq3.'bufrel3.Writer)\{bufacq1,bufacq3,bufrel1,bufrel3}
5: (Buffers1 | 1.Thinker1 | Thinker0 |
$'bufacq3.'bufrel3.Writer)\{bufacq1,bufacq3,bufrel1,bufrel3}
6: (($bufacq1.Buffers0 + $bufrel1.Buffers2) | Thinker1 | 2.Thinker1 |
$'bufacq3.'bufrel3.Writer)\{bufacq1,bufacq3,bufrel1,bufrel3}
```

Figure 6.21

Model

```
bi Reader0 'bufacq1.Thinker0
bi Thinker0 3.Thinker1
bi Thinker1 'bufrel1.Reader1
bi Reader1 1.Reader0
bi Writer $'bufacq3.'bufrel3.Writer
bi Buffers3 $bufacq1.Buffers2 + $bufacq3.Buffers0
bi Buffers2 $bufacq1.Buffers1 + $bufrel1.Buffers3
bi Buffers1 $bufacq1.Buffers0 + $bufrel1.Buffers2
bi Buffers0 $bufrel1.Buffers1 + $bufrel3.Buffers3
bi Reader0a 'bufacq1.Thinker0
bi Thinker0a Thinker1
bi Thinker1a 'bufrel1.Reader1
bi Reader1a Reader0
bi Modela\(Buffers3|Reader0a|Reader0a|Writer)\{bufacq1,bufacq3,bufrel1,bufrel3}
```

Output

```
Command: states Modela
1: Modela = (Buffers3 | Reader0a | Reader0a | Writer ) \{bufacq1,bufacq3,bufrel1,bufrel3}
2: (Buffers0 | Reader0a | Reader0a | 'bufrel3.Writer) \{bufacq1,bufacq3,bufrel1,bufrel3}
3: (Buffers2 | Reader0a | Thinker0 |Writer) \{bufacq1,bufacq3,bufrel1,bufrel3}
  = (Buffers2| Thinker0| Reader0a | Writer) \{bufacq1,bufacq3,bufrel1,bufrel3}
  = (Buffers2| Reader0 | Thinker0 | Writer) \{bufacq1,bufacq3,bufrel1,bufrel3}
  = (Buffers2| Thinker0| Reader0 | Writer) \{bufacq1,bufacq3,bufrel1,bufrel3}
4: (Buffers1 | Thinker0 | Thinker0 | Writer)\{bufacq1,bufacq3,bufrel1,bufrel3}
      = (Buffers1 | Thinker0 | Thinker0 | $'bufacq3.'bufrel3.Writer)
\{bufacq1,bufacq3,bufrel1,bufrel3}
5: (($bufacq1.Buffers0 +$bufrel1.Buffers2)| 2.Thinker1| 2.Thinker1| $'bufacq3.'bufrel3.Writer)
\{bufacq1,bufacq3,bufrel1,bufrel3}
6: (($bufacq1.Buffers0 + $bufrel1.Buffers2) | 1.Thinker1 | 1.Thinker1 |
$'bufacq3.'bufrel3.Writer)\{bufacq1,bufacq3,bufrel1,bufrel3}
7: (($bufacq1.Buffers0 + $bufrel1.Buffers2) | Thinker1 | Thinker1 |
$'bufacq3.'bufrel3.Writer)\{bufacq1,bufacq3,bufrel1,bufrel3}
8: (Buffers2 | Reader1 | Thinker1 | $'bufacq3.'bufrel3.Writer) \{bufacq1,bufacq3,bufrel1,bufrel3}
      = (Buffers2 | Thinker1 | Reader1 | $'bufacq3.'bufrel3.Writer)
\{bufacq1,bufacq3,bufrel1,bufrel3}
9: (Buffers3 | Reader1 | Reader1 | $'bufacq3.'bufrel3.Writer) \{bufacq1,bufacq3,bufrel1,bufrel3}
  = (Buffers3 | Reader1 | Reader1 | Writer) \{bufacq1,bufacq3,bufrel1,bufrel3}
10: (Buffers0 | Reader1 | Reader1 | 'bufrel3.Writer)\{bufacq1,bufacq3,bufrel1,bufrel3}
11: (($bufacq1.Buffers2 + $bufacq3.Buffers0) | Reader0 | Reader0 | $'bufacq3.'bufrel3.Writer)
\{bufacq1,bufacq3,bufrel1,bufrel3}
12(Buffers0|Reader0|Reader0|'bufrel3.Writer)\{bufacq1,bufacq3,bufrel1,bufrel3}
13: (Buffers2 | Reader0 | Thinker0 | $'bufacq3.'bufrel3.Writer)
\{bufacq1,bufacq3,bufrel1,bufrel3}
      = (Buffers2 | Thinker0 | Reader0 | $'bufacq3.'bufrel3.Writer)
\{bufacq1,bufacq3,bufrel1,bufrel3}
14: (Buffers3 | Reader0 | Reader0 | Writer)\{bufacq1,bufacq3,bufrel1,bufrel3}
```

Figure 6.22

Model

```
bi B1 'ja1.B2
bi B2 'tr2.B3
bi B3 'ta1.B4
bi B4 'tr1.B5
bi B5 'jr1.0
bi Tugs2 (ta1.Tugs1)+(ta2.Tugs0)
bi Tugs1 (ta1.Tugs0)+(tr1.Tugs2)
bi Tugs0 (tr1.Tugs1)+(tr2.Tugs2)
bi Jetty2 (ja1.Jetty1)
bi Jetty1 (ja1.Jetty0) + (jr1.Jetty2)
bi Jetty0 (jr1.Jetty1)
bi Model (Tugs2 | Jetty2 | B0 | B0 | B0)\{ta1,ta2,tr1,tr2,ja1,jr1}
```

Output

```
Command: fdobs Model
===  ===> Model                                                                  *
===  ===> (Tugs0 | Jetty2 | B0 | B0 | B1)\{ja1,jr1,ta1,ta2,tr1,tr2}            *
===  ===> (Tugs0 | Jetty1 | B0 | B0 | B2)\{ja1,jr1,ta1,ta2,tr1,tr2}            *
===  ===> (Tugs2 | Jetty1 | B0 | B0 | B3)\{ja1,jr1,ta1,ta2,tr1,tr2}            *
===  ===> (Tugs0 | Jetty1 | B0 | B1 | B3)\{ja1,jr1,ta1,ta2,tr1,tr2}            *
===  ===> (Tugs1 | Jetty1 | B0 | B0 | B4)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty0 | B0 | B2 | B3)\{ja1,jr1,ta1,ta2,tr1,tr2}            *
===  ===> (Tugs2 | Jetty1 | B0 | B0 | B5)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty1 | B0 | B1 | B5)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs2 | Jetty0 | B0 | B3 | B3)\{ja1,jr1,ta1,ta2,tr1,tr2}            *
===  ===> (Tugs2 | Jetty2 | B0 | B0 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty0 | B0 | B2 | B5)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty0 | B1 | B3 | B3)\{ja1,jr1,ta1,ta2,tr1,tr2}            **
===  ===> (Tugs0 | Jetty2 | B0 | B1 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty0 | B0 | B3 | B4)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty0 | B0 | B4 | B4)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty1 | B0 | B2 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs2 | Jetty0 | B0 | B3 | B5)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty0 | B1 | B3 | B5)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty0 | B0 | B4 | B5)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs2 | Jetty1 | B0 | B3 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty1 | B1 | B3 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty1 | B0 | B4 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs2 | Jetty0 | B0 | B5 | B5)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty0 | B1 | B5 | B5)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty0 | B2 | B3 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs2 | Jetty1 | B0 | B5 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty1 | B1 | B5 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs2 | Jetty0 | B3 | B3 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs2 | Jetty2 | B0 |  0 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty0 | B2 | B5 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty2 | B1 |  0 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty0 | B3 | B4 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty0 | B4 | B4 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty1 | B2 |  0 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs2 | Jetty0 | B3 | B5 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty0 | B4 | B5 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs2 | Jetty1 | B3 |  0 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty1 | B4 |  0 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs2 | Jetty1 | B5 |  0 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
```

Figure 6.25

Model

```
bi B0 'ta2.B1
bi B1 'ja1.B2
bi B2 'tr2.B3
bi B3 'ta1.B4
bi B4 'tr1.B5
bi B5 'jr1.0

bi Tugs3 (ta1.Tugs2)+(ta2.Tugs1)
bi Tugs2 (ta1.Tugs1)+(ta2.Tugs0)+(tr1.Tugs3)
bi Tugs1 (ta1.Tugs0)+(tr1.Tugs2)+(tr2.Tugs3)
bi Tugs0 (tr1.Tugs1)+(tr2.Tugs2)

bi Jetty2 (ja1.Jetty1)
bi Jetty1 (ja1.Jetty0) + (jr1.Jetty2)
bi Jetty0 (jr1.Jetty1)

bi Model (Tugs3 | Jetty2 | B0 | B0 | B0)\{ta1,ta2,tr1,tr2,ja1,jr1}
```

Output

```
Command: fdobs Model
===  ===> Model
===  ===> (Tugs1 | Jetty2 | B0 | B0 | B1)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty1 | B0 | B0 | B2)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs3 | Jetty1 | B0 | B0 | B3)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty1 | B0 | B1 | B3)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs2 | Jetty1 | B0 | B0 | B4)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty1 | B0 | B1 | B4)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty0 | B0 | B2 | B3)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs3 | Jetty1 | B0 | B0 | B5)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty0 | B0 | B2 | B4)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty1 | B0 | B1 | B5)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs3 | Jetty0 | B0 | B3 | B3)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs3 | Jetty2 | B0 | B0 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty0 | B0 | B2 | B5)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty0 | B1 | B3 | B3)\{ja1,jr1,ta1,ta2,tr1,tr2}  *
===  ===> (Tugs1 | Jetty2 | B0 | B1 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs2 | Jetty0 | B0 | B3 | B4)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty0 | B1 | B3 | B4)\{ja1,jr1,ta1,ta2,tr1,tr2}  *
===  ===> (Tugs1 | Jetty0 | B0 | B4 | B4)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty1 | B0 | B2 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs3 | Jetty0 | B0 | B3 | B5)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty0 | B1 | B3 | B5)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs2 | Jetty0 | B0 | B4 | B5)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs3 | Jetty1 | B0 | B3 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty0 | B1 | B4 | B5)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty1 | B1 | B3 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs2 | Jetty1 | B0 | B4 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs3 | Jetty0 | B0 | B5 | B5)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty1 | B1 | B4 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty0 | B1 | B5 | B5)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty0 | B2 | B3 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs3 | Jetty1 | B0 | B5 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs0 | Jetty0 | B2 | B4 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty1 | B1 | B5 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs3 | Jetty0 | B3 | B3 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs3 | Jetty2 | B0 |  0 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty0 | B2 | B5 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty2 | B1 |  0 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs2 | Jetty0 | B3 | B4 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty0 | B4 | B4 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs1 | Jetty1 | B2 |  0 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs3 | Jetty0 | B3 | B5 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs2 | Jetty0 | B4 | B5 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs3 | Jetty1 | B3 |  0 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs2 | Jetty1 | B4 |  0 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs3 | Jetty1 | B5 |  0 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
===  ===> (Tugs3 | Jetty2 |  0 |  0 |  0)\{ja1,jr1,ta1,ta2,tr1,tr2}
```

Figure 6.27

Model

```
bi Arrival 'cbAdd1.Arrival
bi WrapMC 'cbRem1.Wrapping
bi Wrapping oBuffAdd1.WrapMC
bi AGVShuttle 'oBuffRem1.AGVShuttle
bi OBuff1 oBuffRem1.OBuff0
bi OBuff0 oBuffAdd1.OBuff1
bi CBelt0 cbAdd1.CBelt1
bi CBelt1 cbAdd1.CBelt2 + cbRem1.CBelt0
bi CBelt2 cbAdd1.CBelt3 + cbRem1.CBelt1
bi CBelt3 cbAdd1.CBelt4 + cbRem1.CBelt2
bi CBelt4 cbRem1.CBelt3

bi Model (Arrival | WrapMC | AGVShuttle | OBuff1 | CBelt0)\
\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}

bi Model2 (Arrival | WrapMC | OBuff1 | CBelt0)\
\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
```

Output

```
Command: states Model2
1: Model2
2: (Arrival | WrapMC | OBuff1 | CBelt1)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
3: (Arrival | WrapMC | OBuff1 | CBelt2)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
4: (Arrival | Wrapping | OBuff1 | CBelt0)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
5: (Arrival | WrapMC | OBuff1 | CBelt3)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
6: (Arrival | Wrapping | OBuff1 | CBelt1)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
7: (Arrival | WrapMC | OBuff1 | CBelt4)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
  = (Arrival | Wrapping | OBuff1 | CBelt2)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
8: (Arrival | Wrapping | OBuff1 | CBelt3)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
9: (Arrival | Wrapping | OBuff1 | CBelt4)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}

Command: states Model
1: Model
2: (Arrival | WrapMC | AGVShuttle | OBuff0 | CBelt0)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
3: (Arrival | WrapMC | AGVShuttle | OBuff1 | CBelt1)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
4: (Arrival | WrapMC | AGVShuttle | OBuff0 | CBelt1)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
5: (Arrival | WrapMC | AGVShuttle | OBuff1 | CBelt2)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
6: (Arrival | Wrapping | AGVShuttle | OBuff1 | CBelt0)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
7: (Arrival | WrapMC | AGVShuttle | OBuff0 | CBelt2)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
8: (Arrival | Wrapping | AGVShuttle | OBuff0 | CBelt0)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
9: (Arrival | WrapMC | AGVShuttle | OBuff1 | CBelt3)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
10: (Arrival | Wrapping | AGVShuttle | OBuff1 | CBelt1)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
11: (Arrival | WrapMC | AGVShuttle | OBuff0 | CBelt3)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
12: (Arrival | Wrapping | AGVShuttle | OBuff0 | CBelt1)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
13: (Arrival | WrapMC | AGVShuttle | OBuff1 | CBelt4)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
14: (Arrival | Wrapping | AGVShuttle | OBuff1 | CBelt2)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
15: (Arrival | WrapMC | AGVShuttle | OBuff0 | CBelt4)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
      = (Arrival | Wrapping | AGVShuttle | OBuff0 | CBelt2)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
16: (Arrival | Wrapping | AGVShuttle | OBuff1 | CBelt3)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
17: (Arrival | Wrapping | AGVShuttle | OBuff0 | CBelt3)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
      = (Arrival | Wrapping | AGVShuttle | OBuff1 | CBelt4)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
18: (Arrival | Wrapping | AGVShuttle | OBuff0 | CBelt4)\{cbAdd1,cbRem1,oBuffAdd1,oBuffRem1}
```