answer to this question is simply the reduced cost of studying a system via simulation as opposed to building a prototype. The time required to implement a simulation is much less than the time required to build a prototype of a distributed or multitasking system. A simulation model can also be executed using virtual time (eg. message transmissions are not really subject to transmission delays) which can reduce the actual time required for an experimental run. Another important reason for using simulation is that it provides complete control over input parameters and allows for the collection of performance information which may be very difficult to acquire in a real situation.

MTS was developed as a design tool for concurrent systems and algorithms as well as to support the validation of analytical performance models. In order to allow a wide variety of systems to be studied, MTS has been made as flexible as possible with respect to hardware and software architectures. MTS models describe both the hardware and software components of a system in detail. Flexible built-in abstractions are provided for modeling processors and interprocessor communication mechanisms. The resulting design tool allows simulation experimentors to study the effects of varying processing and/or communication delays along with other system parameters.

MTS uses software component classes along with dynamically tracked family hierarchies of software components to support both static and dynamic software components within a simulation. Static software components are created at the beginning of a simulation and exist throughout the entire experiment while dynamic software components are created and discarded as required during the simulation. Family trees, identifying dynamically created tasks with their creators, are automatically maintained by the system. The use of dynamic as well as static software components enhances the ability of MTS to model a wide range of concurrent systems.

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

Graphically Defining Simulation Models of Concurrent Systems

by

H. Glenn Brauen, B. Sc. (CS)

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

July 7, 1988

© copyright

1988, H. Glenn Brauen

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-46283-3

The undersigned recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis
"GRAPHICALLY DEFINING SIMULATION MODELS OF CONCURRENT SYSTEMS"
submitted by H. Glenn Brauen, B. Sc. (CS)
in partial fulfilment of the requirements for
the degree of Master of Computer Science

Thesis Supervisor

Chairman, School of Computer Science

Carleton University

date

Abstract

MTS (MultiTasking Simulator), a discrete-event simulation system, has been developed as a design tool for concurrent systems and algorithms as well as to support the validation of analytical performance models. The system is based on a model of Communicating State Machines (CSMs) and supports detailed modeling of the hardware and software components of a system. Built-in abstractions are provided for simulating processors and interprocessor communication mechanisms.

MTS has been developed as an hybrid - run-time simulation models are described as C programs while a graphical user interface for model definition has been developed using Smalltalk-80. The graphical user interface developed for MTS supports modular model development, component abstraction using a clustering mechanism and encourages the development and use of a library of models and model components. Experience gained with the system, suggests that the user interface is adequate for studying large and complex systems.

Using MTS, a study of distributed mutual exclusion on arbitrary, connected networks has been done. A new algorithm is presented which has lower bit complexity, for most network topologies, than the only other algorithm for distributed mutual exclusion on an arbitrary network known to the author.

Acknowledgements

I would like to thank my supervisor, Professor John Neilson, for the many hours spent discussing this work. The many suggestions and questions which arose during these discussions have greatly improved the work presented here.

MTS existed, in an earlier version, prior to my beginning work on this thesis. The original version of the MTS simulation engine was developed by Dorina Petriu and Professor John Neilson.

Finally, I would like to thank my wife, Grace, for the support and encouragement she has provided throughout my studies at Carleton and my son, Patrick, for helping me to wake up in the mornings.

Table of Contents

	Page
ACCEPTANCE SHEET	ii
ABSTRACT	iii
ACKNOWLEDGEMENTS	iv
TABLE OF CONTENTS	v
LIST OF TABLES	vii
LIST OF FIGURES	viii
 CHAPTER 1: Introduction	 1
CHAPTER 2: A Survey of Graphical Specification Languages and Systems	6
2.1 STILE - STructure Interconnection Language and Environment	7
2.2 Statecharts	9
2.3 Predicate/Transition Nets and SPECS	12
2.4 Communicating State Machines	15
 CHAPTER 3: Overview of MTS	 20
3.1 The Modified Communicating State Machine Model	22
3.2 Basic State Machine Interpretation in MTS	23
3.3 Overview of MTS Model Definition Interface Concepts	25
3.3.1 Static Task/Alias Diagrams	28
3.3.2 Dynamic Task/Alias Diagrams	29
3.3.3 Hierarchically Structured Graphs Using Clusters and Shadows	31
3.3.4 Some Smalltalk-80 User Interface Basics	34
3.4 Built-in Modeling Abstractions	35
3.4.1 Processors and Processor Pools	36
3.4.2 Communication Network	42
3.5 Collecting Statistic Observations in MTS	47
3.6 Summary	50
 CHAPTER 4: Class Behaviour Definition in MTS	 52
4.1 Editing State and Transition Parameters	58
4.1.1 Transition Parameter Editing	58
4.1.2 Cluster State Parameter Editing	60
4.1.3 Connect State Parameter Editing	61
4.1.4 Delay and Process State Parameter Editing	62
4.1.5 Receive State Parameter Editing	63
4.2 State Machine Yellow Button Editing Operations	64
4.3 Summary	65

CHAPTER 5: Model Definition in MTS	67
5.1 Static Task/Alias Yellow Button Menu	73
5.2 Class List Yellow Button Menu	74
5.3 Editing Dynamic Task/Alias Information	76
5.4 Editing Processor Pool Information	78
5.5 Editing Processor Pool Parameters	79
5.6 Editing Communication Network Parameters	80
5.7 Editing Model Run-time Parameters	82
5.8 Summary	83
CHAPTER 6: A Case Study of Distributed Mutual Exclusion	85
6.1 The HPR Algorithm	87
6.2 MTS Implementation of the HPR Algorithm	90
6.3 A New Distributed Algorithm for Mutual Exclusion	100
6.3.1 Discussion of the New Algorithm	105
6.4 MTS Implementation of the New Algorithm	107
6.5 Simulation Parameters and Results	117
6.6 Comparison of Asymptotic Message and Bit Complexities	125
6.7 Summary	126
CHAPTER 7: Conclusion and Suggestions for Future Research	128
REFERENCES	131
APPENDIX A: Format of Generated Source Files	135
APPENDIX B: Graphical Editing Modes and Operations	141
B.1 State Machine Editing Modes in MTS	141
B.1.1 "add arc/state" mode	141
B.1.2 "add arc using labels" mode	142
B.1.3 "delete arc/state" mode	142
B.1.4 "move arc/state" mode	143
B.1.5 "cluster states" mode	143
B.1.6 "expand a cluster" mode	144
B.1.7 "move arc point" mode	144
B.1.8 "set arc/state parms" mode	144
B.2 Static Task/Alias Diagram Editing Modes in MTS	145
B.2.1 "add task/alias" mode	145
B.2.2 "add alias using labels" mode	146
B.2.3 "delete task/alias" mode	146
B.2.4 "move task/alias" mode	147
B.2.5 "cluster tasks" mode	147
B.2.6 "expand a cluster" mode	147
B.2.7 "move alias point" mode	148
B.2.8 "set task/alias parms" mode	148

List of Tables

Table	Description	Page
6.1	Table of message transmission times	115
6.2	Minimum average delay estimates for the test networks with each algorithm	121
6.3	Longest path minimum transmission time for HGB algorithm in the tree network	122
6.4	Asymptotic bit and message complexities	124

List of Figures

Figure	Description	Page
2.1	STILE Graph example	6
2.2	A simple STILE box to replace the FORK-MULTIPLY pair	7
2.3	Statechart example	8
2.4	Statechart for Squarer example	10
2.5	Petri net example	11
2.6	Predicate/Transition net for the Squarer example	13
2.7	Squarer example using the SPECS abstraction facility	14
2.8	CSM representation of the Squarer example	16
2.9	Task Messaging Diagram for Squarer example	17
3.1	MTS model components	21
3.2	MTS event handling extensions to the CSM model	22
3.3	Basic model interpretation in MTS	23
3.4	The MTS model development process	26
3.5	Task/Alias diagram for Fork-Multiply system	27
3.6	Task/Alias diagram using MTS collection support	28
3.7	Shadowing and Clustering	31
3.8	A state diagram structured hierarchically using Shadows and Clusters	32
3.9	A hierarchically structured task/alias diagram for the Squarer example	32
3.10	Smalltalk-80 window example	33
3.11	Two Smalltalk-80 soft buttons	34
3.12	User, processor and timer task relationships	35
3.13	CSM segment to request processing	36
3.14	Possible messaging scenario for a single processing request	37
3.15	CSM description of a processor task	38
3.16	CSM descriptions of the timer task	38
3.17	Relationship of user tasks to the communication network	42
3.18	CSM description of the communication network manager	44
3.19	CSM description of the communication timer	45
4.1	* An instance of MtsStateMachineEditor viewing DPCProducer	51
4.2	The remainder of the DPCProducer state diagram	52
4.3	Some simple editing operations	55
4.4	MTS transition parameter editor	57
4.5	MTS cluster state parameter editor	59
4.6	MTS connect state parameter editor	60
4.7	MTS delay state parameter editor	61
4.8	MTS receive state parameter editor	62
4.9	MTS state machine graphical view yellow button menu	62

5.1	The main MTS model editor viewing the definition of DPC	67
5.2	MTS task parameter editor	70
5.3	Static task/alias yellow button menu	71
5.4	Model editor class list yellow button menu	73
5.5	MTS task class dynamic configuration editor	74
5.6	MTS processor pool editor	77
5.7	MTS pool parameter editor	78
5.8	MTS communication parameter editor	79
5.9	MTS run-time parameter editor	80
6.1	Node structure for the MTS implementation of the HPR algorithm	88
6.2	HPRInvoker task class functional description	90
6.3	HPRTransmitter task class functional description	91
6.4	HPRRouter (level 0)	92
6.5	HPRRouter/handle incoming traffic (level 1)	93
6.6	HPRRouter/handle incoming traffic/incoming request (level 2)	95
6.7	HPRRouter/handle incoming traffic/incoming token (level 2)	96
6.8	HPRRouter/handle outgoing traffic (level 1)	97
6.10	Node structure for the MTS implementation of the HGB algorithm	106
6.11	HGBInvoker task class functional description	107
6.12	HGBRepReceiver task class functional description	108
6.13	HGBRouter (level 0)	109
6.14	HGBRouter/handle incoming traffic (level 1)	110
6.15	HGBRouter/handle incoming traffic/handle REQUEST (level 2)	112
6.16	HGBRouter/handle outgoing traffic (level 1)	113
6.17	HGBRouter/handle outgoing traffic/outgoing REQUEST (level 2)	114
6.18	HGBRouter/handle outgoing traffic/send deferred REPLYs (level 2)	114
6.19	The tree and general network maps	116
6.20	The tree network graphs	118
6.21	The complete network graphs	119
6.22	The general network graphs	120
6.23	Low traffic estimate matrices for tree network using the HPR algorithm	122
A.1	Generated source code file layout	133
A.2	Sender class functional definition	134
A.3	Receiver class functional definition	134
A.4	Static task/alias diagram for the model	134

Chapter 1

Introduction

Multitasking and distributed systems are becoming more common as the cost of hardware decreases. The current trend in office environments is to distribute computing power according to need and to provide file handling and output services using communication networks. At the same time, decreasing microprocessor costs are encouraging system designers to use available components to build multiprocessor systems for many applications such as real-time processing. A common feature in these systems is the need to control the concurrent execution of multiple processing units using the available communication mechanism, whether that is a bus or a network.

The goals of distribution vary from one application to the next and are often difficult to achieve in practice. The stated set of goals usually includes the following:

- 1) *improved system performance.* Through concurrent execution, it is often possible to improve system throughput or response time for a given application.
- 2) *increased system flexibility and extensibility.* In general, it is a simpler matter to reconfigure a distributed system, either by moving processing units or communication mechanisms within the system or by adding additional hardware to the system, than is the case with a centralized system.
- 3) *fault tolerance.* By distributing computing power, the problems associated with down-time on a centralized system may be avoided. Depending on system design, loss of one component of a distributed system should either affect only the activities which required the failed part or affect the entire system only slightly.

The performance of distributed systems has often been lower than the original goals of the designers. Part of the reason for this can be found in the extra complexity introduced to the design problem when software considerations so directly affect the performance of a system, as in the case of concurrent systems. The pattern of communications between hardware elements in a concurrent system can directly affect the utilization of system components and the overall performance of the system. In most cases, the communication patterns of these systems are controlled directly by software. Analyzing the effects of component interactions (hardware and software components) on overall system performance is a very difficult task. Furthermore, while it is often a straightforward matter to extend or reconfigure a distributed system, the performance implications of modifying an existing system are not always clear.

Some problems are encountered in the design and implementation of concurrent systems which do not arise when dealing with many centralized systems. The specification and implementation of concurrent systems must deal explicitly with problems of synchronization and control which simply do not arise in standard sequential processing systems. These problems (and their solutions) will often have implications for system performance, as well.

The trend toward distribution does not appear to be lessening in the face of these problems. Indeed, research into the area has increased greatly over the past decade. The result is that new algorithms and architectures are being introduced and studied at a growing rate. The subject of this thesis is a simulation system, called MTS (MultiTasking Simulator), which has been used to study multitasking and distributed systems.

Why would simulation be a useful means of studying concurrent systems? The main

answer to this question is simply the reduced cost of studying a system via simulation as opposed to building a prototype. The time required to implement a simulation is much less than the time required to build a prototype of a distributed or multitasking system. A simulation model can also be executed using virtual time (eg. message transmissions are not really subject to transmission delays) which can reduce the actual time required for an experimental run. Another important reason for using simulation is that it provides complete control over input parameters and allows for the collection of performance information which may be very difficult to acquire in a real situation.

MTS was developed as a design tool for concurrent systems and algorithms as well as to support the validation of analytical performance models. In order to allow a wide variety of systems to be studied, MTS has been made as flexible as possible with respect to hardware and software architectures. MTS models describe both the hardware and software components of a system in detail. Flexible built-in abstractions are provided for modeling processors and interprocessor communication mechanisms. The resulting design tool allows simulation experimentors to study the effects of varying processing and/or communication delays along with other system parameters.

MTS uses software component classes along with dynamically tracked family hierarchies of software components to support both static and dynamic software components within a simulation. Static software components are created at the beginning of a simulation and exist throughout the entire experiment while dynamic software components are created and discarded as required during the simulation. Family trees, identifying dynamically created tasks with their creators, are automatically maintained by the system. The use of dynamic as well as static software components enhances the ability of MTS to model a wide range of concurrent systems.

Although MTS has been described as a design tool for concurrent systems, it is important that the system be capable of providing useful statistical information concerning the performance of systems being studied. For this reason, a programming language which would enable efficient computation was required for the simulation system and C was chosen. An MTS model is encoded as a C program which is then compiled and linked with the rest of the simulator to create the simulation engine. Due to the complexity of the systems being studied, the process of directly coding a model in C is both time consuming and error prone. This thesis presents an environment for model development which enhances the simulation system; enabling models to be developed in shorter periods of time and reducing the number of errors made during the model development process. The graphical user interface developed for MTS supports modular model development and component abstraction using a clustering mechanism. Using the system, a model designer can easily build and access a library of models and model components. Experience gained with the system, suggests that the user interface is adequate for studying large and complex systems.

The design of MTS has been based on a Communicating State Machine (CSM) model, which is an extension of the Finite State Machine (FSM) model. FSMs are well-known to Computer Scientists and have a simple graphical representation which is easily extended to the CSM model. In MTS, the graphical representation of a CSM is used to provide the functional description of a model component. The functional description is annotated with information related to the details of model implementation to complete the specification of the component. Along with the graphical interface for specifying model components, a graphical interface has been developed to support the definition of models using the previously defined components. A graphical user interface developed specifically for the purpose of creating MTS models also has the advantage of presenting the user with only those options which are appropriate at any given time.

As stated above, C was chosen as the development language for the simulator because it was important that the simulator be an efficient program. C, however, is not the most appropriate choice for the development of a graphical user interface. The graphical user interface has been developed in Smalltalk-80, which is an excellent environment for both developing and using a graphical editing interface.

Chapter 2 of this thesis presents a brief survey of graphical languages and systems which have been used to implement, study or specify concurrent systems. Chapter 3 gives an overview of the main concepts used in the development of MTS and the model development interface. Chapters 4 and 5 present the model development interface. Chapter 6 describes a study of distributed mutual exclusion on arbitrary, connected networks. A new algorithm is presented which has lower bit complexity, for most network topologies, than the only other algorithm for distributed mutual exclusion on an arbitrary network known to the author. Chapter 7 presents some conclusions concerning this thesis and discusses possibilities for further research.

Chapter 2

A Survey of Graphical Specification Languages and Systems

Graphical user interfaces have been used extensively to aid people in the definition and solution of problems from a wide variety of domains. Sketchpad [20] was an interactive graphics system which allowed users to edit constrained line diagrams using a computer. ThingLab [1], which extended the ideas in Sketchpad, was a graphical constraint-based simulation kit for solving problems in physics and geometry and, later [2], allowed the constraints on objects to be depicted graphically as well. Graphical programming systems to allow users to develop and/or debug software using representations which fit the problem domain of interest have been studied by several researchers ([6], [11], [13], [16], [17]).

The focus of this chapter is on graphical languages and systems which have been used to implement, study or specify systems which exhibit concurrency.

2.1 STILE - STructure Interconnection Language and Environment

STILE [22] was designed to provide support for the development and interconnection of concurrent software components and was specifically intended for real-time applications. One of the main motivations of STILE was to support the development of reusable software components using a modular approach to system design. The virtual machine for the STILE language is called a *STILE graph*. A STILE graph is composed of a set of *boxes* interconnected by undirected *links*. The links connect to the boxes at points called *ports* which are known only to the box on which they reside and a link may only connect two ports. The boxes each represent either a concurrent process or a subsystem composed of concurrent processes. The ports and links represent devices for interprocess communication.

Figure 2.1: STILE Graph example.

Figure 2.1 shows a component of a STILE graph which would compute the square of an input value. Notice that the port names, "in", "out1" and "out2", are known only to Fork while the port names, "x", "y" and "z", are known only to Multiply. The links are also unknown to the processes - in fact, the links are resolved prior to run-time by combining linked ports as a single entity. Since each box has knowledge concerning only its local ports, each box must be created as an independent module. No assumptions or information concerning the other processes is utilized within each module. Each process functions by testing the state of its local ports and carrying out actions based on the result of the tests.

As mentioned above, a box in a STILE graph can represent either a simple process or an arbitrary subgraph. Since port names are only known locally, a box representing a subgraph is interchangeable with a simple box which performs the same logical function. For instance, the graph component of Figure 2.1 could be replaced with the simple box shown in Figure 2.2.

Figure 2.2: A simple STILE box to replace the FORK-MULTIPLY pair.

The port mechanism used in STILE provides simple and flexible communication between boxes. Each port has an associated *type*, *state*, and *value* where the *type* is the data type of the port's value, the *state* provides information to the box concerning activities regarding the port and *value* is the datum which is currently available at the port. Note that the port does not queue data for the box. Each port can be marked "full" or "empty", allowing a box to test the state of its ports. Thus, a box can determine whether or not a value is waiting on an input port and can determine whether or not a value has been removed from an output port. Using this mechanism, boxes can communicate either synchronously by waiting for the communication partner to set the port state or asynchronously by simply reading and writing the port value as often as required.

The STILE system seems to be an interesting framework for the creation of modular, concurrent systems and for encouraging reuse of existing software components. However, STILE graphs are statically defined and cannot be changed at run-time. The port-link abstraction used at graph construction time does not seem to be extendible to a system in

which run-time task creation would be supported. STILE has been designed to ensure that concurrent components of a system require no knowledge of each other, therefore it is unclear that providing processes with the information required to create other processes would be either straightforward or desirable.

2.2 Statecharts

Statecharts [8] are an extension of conventional state diagrams which incorporate state clustering, inter-level state transitions and a broadcast mechanism for communication between concurrent components. Simple assumptions concerning the meaning of inter-level transitions and the meaning of clustered states have resulted in a very compact description language which seems to be capable of describing quite complex systems. A complete description of statecharts is beyond the scope of this survey but a simple introduction along with a couple of examples will be presented.

Figure 2.3: Statechart example.

Figure 2.3 shows a simple statechart example. The first thing to notice is the super-state labelled D which contains two states, A and C. States A and C represent an XOR decomposition of state D in the sense that when the system is in state D, it is really in one

of A or C and not both. State transitions are shown using directed arcs which are labelled using the format *event[(condition)]/[action]*. Assuming there is no optional condition specified, the transition will be traversed when the specified event occurs while the system is in the state from which the transition originates. If a condition has been specified for the transition then it will be traversed only if the condition is true. If an optional action has been specified for the transition then the action occurs as part of the traversal process; this action may, in turn, be an event triggering another transition in the system. The transition labelled "w" is an inter-level transition - it indicates that if the system is in state D (ie. either state A or C) when event "w" occurs the system will enter state B. In other words, the arc labelled "w" represents a transition from any of D's substates to B. The pointers to states D and A (top-left of each state's outline) indicate default states for the system - the pointer to state D indicates that state D is the default state for the system while the pointer to state A indicates that state A is the default state among D's substates. Combining these pointers, we see that state A is the true default state for the system.

Figure 2.4 shows a statechart for a system which squares an input value using two concurrent components - one to replicate the input and one to multiply the replicated inputs together thereby producing the result. SQUARER is an example of a new type of super-state used in statecharts in which the substates (FORK and MULTIPLY) form an AND decomposition of the super-state. This means that being in state SQUARER implies being in state FORK and state MULTIPLY simultaneously, each of which are also super-states. Therefore, the AND decomposition is used to describe concurrency in statecharts. By default, the FORK component of the SQUARER is in the state labelled "wait for input" and the MULTIPLY component is in the state labelled "no inputs set". When the "haveInput" event is signalled, the FORK component traverses the transition causing the actions, "setX" and "setY", which in turn act as events causing transitions in the MULTIPLY component.

Figure 2.4: Statechart for Squarer example.

Statecharts make use of the location of components to reduce the complexity of diagrams compared to conventional state diagrams. State containment is an important feature of statecharts. Using the area of a diagram allows relatively simple diagrams to describe the behaviour of complex systems.

Statecharts were designed to provide behavioural descriptions of complex systems; they do not describe the intended implementation of a system. Consequently, there is no support for the creation of multiple instances of a given type of concurrent component nor is there any support for the dynamic creation of concurrent components. Furthermore, [8] describes events as being communicated using an instantaneous broadcast mechanism. It is unclear what would be the consequences of assuming a more general communication mechanism.

2.3 Predicate/Transition Nets and SPECS

A *Petri net* [15] is a formal model of information flow which describes concurrency very well. A Predicate/Transition net (eg. see [7]) is a Petri net in which global or local data may be modelled and in which actions and conditional tests can be applied to the data. As a result, Predicate/Transition nets provide increased modelling capability but are harder to analyze than basic Petri nets. This section will briefly describe Petri nets, Predicate/Transition nets and SPECS (SPECification of Concurrent Systems) [5] which is a graphical system developed using Predicate/Transition nets.

A Petri net is a model composed of: *transitions* representing events in the model; *places* representing conditions in the model; *tokens* which are used to represent the state of the system and directed arcs which are used to connect transitions to places and vice versa. The transitions, places and arcs model the static properties of a system while the placement and movement of tokens is used to model the dynamic properties of the system.

Figure 2.5: Petri net example.

Figure 2.5 shows the graph representation of a simple Petri net containing two places, A and B, and one transition, T. Notice that, with respect to T, A is called an *input place* while B is called an *output place*. Likewise, T is called an *output transition* with respect to A while it is an *input transition* with respect to B. The presence of a token in place A is indicated by a dot.

As stated above, the dynamic properties of a Petri net are modelled by the movement of tokens. The movement of tokens is, in turn, controlled by the firing of transitions. A transition may fire when each of its input places contain a token. When the transition fires, a token is removed from each input place and a token is placed in each output place.

Figure 2.6 shows the Squarer example as a Predicate/Transition net in which a value is associated with each token. Notice that the Fork and Multiply components are now represented within a single connected net. The Fork component is represented by a single transition with a single input place and two output places - the result of this transition firing is that two copies of the input token are created and moved into each of the output places, copy1 and copy2. The Multiply component is also represented by a single transition which accepts two input tokens, one from each of copy1 and copy2, and generates a single output token on which the value is the product of the two input token values. The labels on the input and output transitions, along with the notation indicating the actions to be performed during the transition, indicate the data transformations which occur as data flows through the net.

Since the Fork and Multiply transitions can work concurrently, depending on the presence of tokens, some assumptions must be made concerning the flow of tokens through the net for this simple Squarer to function properly. Assuming that tokens flow through a place on a first-in-first-out basis, this simple net will work. Otherwise, the net would have to be modified to ensure that copied inputs are paired properly going into the Multiply component.

Figure 2.6: Predicate/Transition net for the Squarer example.

SPECS [5] is a Smalltalk-based system, based on Predicate/Transition nets, for designing distributed systems and protocols. Data is associated with tokens. Actions and conditions on the data are specified using Smalltalk code. The system provides support for creating and editing nets graphically and for simulating and animating the activities of a net. The editing tools support refinement and abstraction functions on the net allowing logical components to be grouped into subnets.

Figure 2.7 shows the Squarer example represented using the SPECS abstraction facility. Figure 2.7(a) shows the high-level depiction of the squarer and Figure 2.7(b) shows the segment of the net which is represented, in Figure 2.7(a), by the rectangle labelled "SQUARER". Notice that the actions associated with the Multiply transition are now written as Smalltalk code and inscribed directly on the transition, represented as a rectangle.

SPECS seems to provide a good set of editing tools for the creation and manipulation of Predicate/Transition nets. However, despite the refinement and abstraction facilities provided with SPECS, reusability of system components is still restricted by the fact that a Petri net model is represented by a single connected graph. The presence of a token in a

Petri net can be used to represent either the presence of a data item or the presence of an active process. This can make it difficult to decompose a Petri net or Predicate/Transition net into logical processes, especially in a system in which the dynamic creation of processes is to be modelled.

Figure 2.7: Squarer example using the SPECS abstraction facility.

2.4 Communicating State Machines

The communicating state machine (CSM) model (eg. see [19]) is an extension of the finite state machine (FSM) model. The main motivation for the extensions to the FSM model was to allow greater modelling power; the cost of the extensions is found in the more limited analysis techniques which may be applied. The CSM model provides for the use of local and/or global data by machines and allows communication to be carried out between machines by message passing. In the case of the CSM model, modelling power is significantly improved but even simple models can be very difficult to analyze.

In general, a state is used to represent the execution of some non-atomic action (eg. message reception or processing of data) while state transitions are considered to be instantaneous. Certain non-blocking functions can be carried out during the transition if it is assumed that these functions require zero time. CSM transitions generally have a condition specifying whether or not the arc should be traversed and a list of atomic actions to be performed as part of the traversal process. The notation on the transitions is of the form, *condition/action list*, where the *condition* represents the test carried out to determine whether or not this transition should be traversed and *action list* represents an ordered list of atomic, instantaneous actions which will be carried out as part of the traversal process. A dot (•) in the place of either a condition or an action list specifies a null condition (the transition will always be selected) or action (the transition has no side effects).

Communication between machines is provided using a non-blocking send primitive and a blocking receive primitive with an optional timeout parameter. Using these primitives, either asynchronous communication or rendezvous-style communication can be implemented. The messaging notation is a variation of that used in CSP [10].

A receive operation is specified using the format, *[sender]?entry[,timeout]*, where *sender* specifies from which machine the receiver will accept messages, *entry* specifies the type or types of messages which will be accepted and *timeout* represents the amount of time the machine is willing to wait before giving up on receiving a message. Everything except *entry* is optional. By not specifying a *sender*, messages will be accepted from any other machine. By not specifying a timeout it is assumed that the machine will wait as long as required to receive a message. Note that, although the receive operation is defined as a blocking operation, a timeout of zero will effectively provide a non-blocking receive primitive.

A non-blocking send operation is specified using the format, *receiver!entry*, where *receiver* specifies the machine to which a message is to be sent and *entry* is the same as in the receive case except that only a single type may be specified. Note that although the send operation is defined as a non-blocking operation, a send/receive operation pair can be used to provide rendezvous communications.

Notice that the "sender?entry" notation can be used to specify a transition condition as well as to specify an action to be performed as part of transition. When used to specify a condition, the notation refers to an action carried out in the transition's originating state.

Figure 2.8: CSM representation of the Squarer example.

Figure 2.8 presents the Squarer example as a pair of CSMs. For this example to work, Fork's variables, output1 and output2, must both be initialized to identify Multiply. Likewise, Multiply's variables, input1 and input2, must be initialized to identify Fork.

To support the reuse of defined CSMs, it is important that machine relationships are not part of the definition of the separate machines. For this reason, a method is required to specify the relationships between members of a set of predefined machines. This is similar to the problem addressed by the graphs used in STILE - there, as here, the definition of a system component was separated from the connection between that component and the other elements of the system. In [3] a graphical design notation is presented for defining the relationships and messaging interactions of a set of Ada tasks. Figure 2.9 shows the Squarer example using this notation. Each task is represented by a parallelogram and directed arcs indicate access relationships between the tasks. The labelled boxes inside the parallelograms are task entries which correspond to the types of message requests accepted by each task. The arrow symbols (all labelled "value", in this case) indicate the flow of data associated with each access relationship. This diagram is incomplete since it does not show where FORK gets the input message from or to what machine MULTIPLY sends the result.

Figure 2.9: Task Messaging Diagram for Squarer example.

CSMs provide a clear and flexible specification notation for multitasking systems, although as the machines get complex the graphical representation can become unwieldy. CSMs,

combined with the notion of machine classes, have been used in the development of a graphical user interface for MTS. The Task Messaging Diagram notation, however, has not been used in MTS because the details usually included in these diagrams seems to be redundant, from a system specification point of view, when a complete CSM definition is available for each machine. For instance, the details concerning the patterns of data flow are specified twice since they must appear in both the pictorial system diagram and in the definitions of the tasks involved. Therefore, while these diagrams are clearly useful as design tools, the decision was made not to use them to graphically specify systems in MTS.

Chapter 3

Overview of MTS

This chapter will present the main concepts used in the development of the user interface and the simulation engine for MTS. As stated earlier, MTS has been designed to aid in the study of multitasking or distributed systems. Therefore, it is important that a model designer be able to easily specify the types of components from which a system is to be built and then specify the details of the system using those components.

Chapter 2 briefly presented the CSM model upon which MTS has been based. It is important to remember that MTS is a simulator while the CSM model is mainly a descriptive tool. In simulation, events must be generated and interpreted in order for the model to be of use. In order to provide an efficient simulation system, some restrictions and enhancements have been made to the basic CSM model. This chapter will describe these modifications and the basic steps involved in interpreting an MTS simulation model. The chapter will also present the main concepts and notations used in the graphical model development environment. This chapter will then describe the built-in modeling abstractions for processor pools and communication mechanisms used in MTS. Finally, the chapter will describe the facilities provided for the collection of statistics in MTS.

Figure 3.1 shows the major components of an MTS model. A *task* is the basic component used in developing an MTS model and generally represents a software process or an independent hardware component. When designing an MTS model, the designer specifies the set of *static tasks* which exist for the entire duration of the simulation. Tasks may create

or kill other tasks during the simulation - these created or killed tasks are called *dynamic tasks* - and family hierarchies (parents and siblings) are automatically tracked by MTS. Each task has local state and a set of tasks interact by exchanging messages using the message primitives described, for CSMs, in Chapter 2.

Figure 3.1: MTS model components.

Every task is an instance of a *task class*. A task class provides the *behavioural definition*, specified as a CSM, for tasks which are instances of the class. A task class also provides model specific information, called the *dynamic configuration information*, concerning the initial state for dynamic tasks which are instances of the class. An MTS model also contains a set of *processor pools*, each of which is a collection of *processors* with common characteristics. A processor is an MTS abstraction which allows processing delays (eg. scheduling, queuing and computation time on a real processor) associated with a task to be simulated. Other elements of an MTS model definition include *communication parameters*,

which specify the type and timing of interprocessor message transmissions, and *run-time parameters*, which specify stopping conditions for the simulation as well as details related to the gathering of data from the simulation. As shown in Figure 3.1, task T is an instance of class C and is allocated to run in processor pool P.

3.1 The Modified Communicating State Machine Model

In order to provide an efficient simulation engine and to simplify event handling, two major changes have been made to the CSM model in the implementation of MTS. The first change is that typed states are used in defining the state machines. The second change is that restrictions have been made on the number, type and order of actions performed as part of a state transition.

The type of a state is one of *connect*, *delay*, *process* or *receive*. A connect state represents a null action - the state is a decision point in the state machine. A delay state's purpose is to cause the task to "sleep" for a specified amount of time. A process state's purpose is to rendezvous with the task's processor until a specified amount of processing time has been simulated for this task. A receive state's purpose is to receive a message - the receiving task will be blocked until either an appropriate message is received or its timeout period (if specified) elapses.

Figure 3.2: MTS event handling extensions to the CSM model.

During a state transition, there are four operations which may be performed; 1) a task may be created, 2) an arbitrary assignment function may be executed, 3) a task may be killed and 4) a message may be sent. All, some or none of these actions may be performed during a specific state transition. Any actions which are performed, however, will be performed in the order stated above. Figure 3.2 depicts the changes to the CSM model.

To create a task, the creator specifies a task class of which the newly created task is to be an instance. The task will be created and the family tree of both the created and creator tasks will be updated to reflect the change. The assignment function may be used to modify or access a task's local data or global data. To kill a task, the killer simply specifies which task is to be killed. The specified task (and any tasks it has created) will be killed and the family tree of the killed task's creator will be updated. To send a message, the sending task must specify the receiver and the type of the message to be sent. The message will be sent and queued until the receiver is in an appropriate state to accept the message.

3.2 Basic State Machine Interpretation in MTS

The basic cycle of state machine interpretation in MTS proceeds as follows:

- 1) *State Entry*: the task enters a state.

When the task enters a state, the type dependent action of the state is initiated. This is generally a blocking action which will account for the passage of time in the simulation. When the state's actions are completed, the task will exit that state.

- 2) *State Exit*: the task exits a state.

When the task exits a state, the task must select an outgoing transition from the state and perform that transition. In order to check a transition's traversal condition each transition has an associated function which returns a boolean value. This value is true if the transition should be performed and false otherwise. The task will execute the arc selection function of each transition until a function returns true, at which time the task will perform the associated transition. In order to simplify the definition of the arc selection functions, the order of arc selection function testing can be explicitly described by the model designer. The order of transition testing at a given state is fixed throughout the simulation. It is an error if all arc selection functions return false; this corresponds to an underspecified state in a finite state machine.

3) *State Transition*: the task performs a state transition.

When the task changes state, the actions associated with that transition are performed (see section 3.1). These actions are assumed to be atomic, instantaneous actions and thus will not account for the passage of time in the simulation. Once these actions are completed, the task will enter the transition's target state.

- 1: state entry
- 2: state exit and transition selection
- 3: transition and associated actions

Figure 3.3: Basic model interpretation in MTS.

Figure 3.3 depicts the basic state machine interpretation process. In MTS, the activities of all tasks are controlled using a single time-ordered event queue. The activities of a task are encoded as events and as each event is removed from the front of the queue, the state of the task associated with the event is updated.

The MTS simulation engine is implemented in C and depends upon user-defined functions to encode the activities of a state machine. The four types of user-defined functions which are required are: 1) arc selection functions, 2) assignment functions, 3) delay functions and 4) process delay functions. Arc selection functions, as described above, are used to determine whether or not the associated state transition should be performed. Assignment functions are optionally executed during state transition to carry out arbitrary activities such as modifying either local or global data. Delay functions are executed when a delay state is entered to determine for how long (in simulated time) the task in question should be suspended. Process delay functions are executed when a process state is entered to determine how much processing time is to be simulated by the executing task's processor on behalf of the task. Each of these functions accepts a single parameter which is a pointer to the executing task's *task control block (tcb)* therefore the task's local state information (instance variables, family hierarchy information, message reception information, etc.) is always available when these functions are being executed.

3.3 Overview of MTS Model Definition Interface Concepts

The preceding sections of this chapter presented an overview of MTS and concepts related to the event handling performed by MTS. This section will describe some of the main ideas which have been used in the development of a graphical user interface for MTS.

The main reason for developing a graphical user interface for MTS was to speed the model

development process by allowing a user to work as directly as possible in the paradigm of CSMs. Another reason was to allow a user to create modular simulation models. In terms of the systems being studied using MTS, the task class definitions correspond to software in the real systems. Thus, the model is composed of a set of tasks, some of which are executing the same "software". From this point of view, it makes sense to allow a user to develop "software" in isolation from the rest of the system for all of the reasons it is supported in the real systems. Another benefit from this approach to MTS model development is that the user can build up a library of both class definitions and models for later use or reference.

The MTS user interface is divided into two parts: the editors which are used to create and modify MTS class behaviour definitions and the editors which are used to create and modify MTS simulation models, using a library of class behaviour definitions as input. Note that the dynamic configuration information for a given class is dependent on the model in which the class is being used and is, therefore, defined specifically for each model.

The first phase of model development is the creation of the class definitions required for a model. The class definitions, once complete, are stored as subclasses of class *MtsStateMachine* in the Smalltalk-80 hierarchy. The second phase of model development is the definition of the simulation model using the previously defined classes as input. Once a model is created, it is stored as a subclass of class *MtsModel* in the Smalltalk-80 hierarchy. At this point, the C language source code may be generated, compiled and linked with the simulator object files to create a simulation engine. The MTS model development process is depicted in Figure 3.4.

Note that the intent of the MTS user interface is not to alleviate the need for the user to

understand the underlying simulation system. Nor is it the intent of the interface design to allow the user to define models without understanding the C programming language. As stated earlier, the simulation engine design assumes the presence of user-defined functions to implement the state machine definition. Therefore, the interface includes a set of text editors, some of which are used to enter C code which will be replicated in the generated source code. The interface is designed to present the user with appropriate options thus helping the user to decide when and where s/he must supply a piece of code to accomplish his/her purpose. The user will have to understand C and the simulator sufficiently to be able to provide the required code segments.

Figure 3.4: The MTS model development process.

3.3.1 Static Task/Alias Diagrams

In this section, the graphical notation used in MTS to describe systems of tasks and the relationships among the tasks will be presented. Every MTS model has an initial set of tasks which is called the *static task set* because these tasks exist for the duration of the simulation. All other tasks which are created during the course of simulation are called *dynamic tasks*. To model a system, the set of static tasks and the inter-relationships of those tasks must be specified. The information concerning the creation of dynamic tasks must be specified in the class definitions for the classes of which the created tasks are instances.

In MTS, *task/alias diagrams* are used to define a system in terms of its static task set and the relationships between the static tasks. Figure 3.5 shows a simple task/alias diagram for the Squarer system presented in Chapter 2. Each task is represented by a rectangle containing the task's name and each alias relationship is represented by a labelled, directed edge. For instance, the edge labelled INPUT1 indicates that MULTIPLY "knows FORK as INPUT1". In terms of the task model used in MTS, this means that MULTIPLY has a local variable called INPUT1 which contains the identifier of FORK.

Figure 3.5: Task/Alias diagram for Fork-Multiply system.

To partially alleviate the need to access arbitrary global data due to the restricted size and structure of a task's local data pool, a collection abstraction has been developed for MTS.

Collections, represented as linked lists of floating-point values, allow a task to manipulate collections of data from the task's assign, delay, process delay or arc selection functions. This collection abstraction can be used, in combination with the task/alias notation, to allow a task to maintain lists of task identifiers which the task views as logically equivalent. Figure 3.6 shows an example task/alias diagram in which a list of aliases is used. In this example, node i knows that each of the nodes, j, k and l, are neighbours in a network. In terms of the task model of MTS, this means that node i has a local variable called NEIGHBOURS which contains the header index of a collection which contains the identifiers of the aliased nodes (in fixed but unspecified order).

Figure 3.6: Task/Alias diagram using MTS collection support.

3.3.2 Dynamic Task/Alias Diagrams

As stated earlier, task classes in MTS are defined in two parts. The first part specifies the behaviour of instances of the class - this is called the *class behaviour definition*. The second component specifies the relationship of dynamically created tasks which are instances of the class to the rest of the statically defined model (static task set and processor pools, etc) - this is called the *dynamic configuration definition*. To specify this

information, MTS uses a restricted version of the task/alias diagram presented in the last section.

In the case of dynamic configuration information, what has to be defined is the relationship of a class prototype to the rest of the statically described system. Notice that this is an easier problem than defining the relationship of a dynamically created task to the system as it existed at time $t-\epsilon$ where t is the time at which the task was created. In order to solve this problem it would be necessary to model the system in a state which is not known at the time the dynamic configuration information is specified! The task/alias diagram which is used for dynamic configuration definitions shows the aliases held by a prototype of a class with respect to members of the static task set (ie. only aliases from the prototype to other tasks are specified).

In effect, what the dynamic configuration information of a class provides is a convenient notation for specifying information which dynamically created instances of a class should have without forcing the creator of the created task to explicitly deliver the information in a message. Of course, forcing the creator to explicitly deliver the information at creation time also allows the creator to deliver information which may have been updated since the static task set was defined.

Note that members of the static task set are not considered to be "dynamically created tasks which were created at time zero". The dynamic configuration information is not applied to the static tasks because to do so would cause problems of circularity when the static task set is created. Since the dynamic configuration information specifies the relationship of class instances to the static task set, applying this information to the static task set would require that the task/alias graph be topologically sorted to determine a creation order for the tasks. Topological sorting implies that the task/alias graph must be cycle free. This restriction on

the relationships between members of the static task set would, for instance, rule out the simple Squarer system. Therefore, there is a distinction made between static and dynamic tasks in MTS.

3.3.3 Hierarchically Structured Graphs Using Clusters and Shadows

In order to control the amount of graphical information presented to the user at a given time and to provide an abstraction mechanism, the related concepts of *clusters* and *shadows* have been used in the MTS user interface. Shadows and clusters are special entities used in the graphical editing tools to allow the state and task/alias diagrams to be structured hierarchically. A *cluster entity* (either a state or a task) is used at level n of an hierarchically structured graph to represent an arbitrary number of objects (either states and state transitions or tasks and aliases) at level $n+1$. A *shadow entity* (a state or state transition in the state diagrams or a task or alias in the task/alias diagrams) is used to represent a single object, from level n , in a subgraph at level $n+1$. Figure 3.7 shows a simple example of clustering and shadowing. Each of the entities adjacent to the cluster in level n and each arc incident on the cluster in level n is represented by a shadow in level $n+1$. Thus, a subgraph at level $n+1$ is represented by a cluster at level n while shadows are used to provide a frame of reference for the level $n+1$ subgraph. Notice, in Figure 3.7, that the arc from A to B in level n is not shadowed in level $n+1$; only arcs incident on the cluster are shadowed.

Figure 3.7: Shadowing and Clustering

In the graphical user interface developed for MTS, the cluster and shadow mechanism has been used to provide a bottom-up abstraction facility. A set of entities (tasks or states) can be created and connected using the editors and then the set of entities can be clustered. This operation moves the elements of the clustered set down one level in the hierarchy and replaces them in the current level by a single cluster entity. Originally, the cluster and shadow mechanism was used to support both a bottom-up and a top-down abstraction facility in the editors, but the resulting mechanisms were found to be clumsy, confusing and inconsistent.

Figure 3.8 shows the structure of a state diagram in which the cluster and shadow abstraction mechanism has been used. Notice that the shadow and cluster mechanism allows a graph to be structured hierarchically to an arbitrary depth since clusters can contain other clusters. Notice also that a unique path can be specified from the highest level of the graph to any node in the graph by listing the names of the clusters entered in order to reach the node.

Figure 3.8: A state diagram structured hierarchically using Shadows and Clusters.

Figure 3.9 shows the task/alias diagram for the Squarer example using the cluster and shadow abstraction mechanism. In this diagram, the rectangle labelled "square" is a cluster representing the fork-multiply task pair in the hierarchy.

Figure 3.9: A hierarchically structured task/alias diagram for the Squarer example.

3.3.4 Some Smalltalk-80 User Interface Basics

The Smalltalk-80 user interface is built using a model of windows. A *window* is a rectangular section of the screen which is used to present information (graphic or textual) to a user concerning some part of the system. Figure 3.10 shows an example of a Smalltalk-80 window. The black tab at the upper left corner of the window is called the *label*. The label is used to provide information concerning the purpose of the window. Only one window in the system can be active at a given time. The label of the active window is displayed with white letters on a black background (*highlighted display*) while all other windows have their labels displayed using black lettering on a white background.

Figure 3.10: Smalltalk-80 window example

Smalltalk-80 supports a mouse with three buttons. The buttons are identified by colours as follows: *red* (leftmost button), *yellow* (middle button), and *blue* (rightmost button). The term "button" is, unfortunately, also used in Smalltalk-80 to designate a graphical representation of a switch which can be in either the "on" position or the "off" position. The buttons on the mouse will be referred to using the appropriate colour designation (ie. red button, yellow button or blue button) while the graphical buttons will be referred to as *soft buttons*. Figure 3.11 shows a pair of soft buttons. A soft button is in the "on"

position if it is highlighted on the display screen and is in the "off" position otherwise. A soft button may be turned "on" or "off" by positioning the cursor on the graphical representation of the soft button and clicking the red button,

Figure 3.11: Two Smalltalk-80 soft buttons

Pop up menus are used extensively in Smalltalk-80 because they are a simple mechanism through which a user may initiate actions or provide necessary information. In the MTS user interface, yellow or blue button menus are used as a method for providing access to operations to the user. In order to access these operations, the user presses the appropriate button (yellow or blue), a menu will be displayed from which an action may be selected and the user selects an operation by positioning the cursor on the desired element of the list and clicking the appropriate mouse button (yellow or blue). The blue button menu used throughout the MTS interface is the standard Smalltalk-80 blue button menu which provides windowing operations such as frame adjustment and window closing, collapsing and deactivation. The yellow button menus are designed to provide application specific operations and will be presented in chapters 3 and 4.

3.4 Built-in Modeling Abstractions

This section will describe the built-in modeling abstractions which have been developed to aid model designers when simulating concurrent systems. The built-in abstractions provided with MTS represent hardware components which are frequently required in simulation models of concurrent systems - namely processors, processor pools and communication mechanisms. These have been developed as built-in abstractions rather than as standard task classes for performance reasons. Initially, these were developed and

tested as task classes being interpreted by the simulator but the additional message traffic was found to significantly impact the performance of the simulator. The abstractions were, therefore, redesigned and coded as part of the simulation engine.

If a model designer decides that the built-in abstractions are not appropriate for a model being designed, it is a simple matter to build an appropriate task class and explicitly implement the type of architecture desired for the model. The earlier versions of MTS required that a model designer develop an explicit task class for processors, for instance, whereas now a designer has the option of using the built-in definition or explicitly overriding the built-in definition with a new class.

3.4.1 Processors and Processor Pools

MTS provides an abstraction for modeling processors and the interaction of tasks with processors. A processor is an abstraction which is used to simulate processing delays for a given task. A task requiring processing rendezvous with the processor long enough to simulate the required amount of processing time. While the processing time is being simulated, the task will remain in a Process state (see section 3.1), thus the time spent in the state will be the total of the requested processing time and the queueing time spent waiting to be serviced at the processor. A processor can function in either preempt/resume or head-of-the-line scheduling mode with or without a timeslicing quantum.

The remainder of this section will present the MTS processor model in terms of CSMs. Note, however, that the processor model is implemented as part of the simulator, so this section is strictly descriptive, showing a possible implementation using communicating state machines. To describe the processor model in terms of CSMs, assume that the processor works in conjunction with a timer task which is used to wake the processor

when a quantum has terminated and to determine the amount of processing time which a task has received prior to an interruption.

Figure 3.12 shows the task/alias diagram representing the relationship of a user task, the processor task and the timer task which is associated with the processor. The aliases shown will be used throughout this section in the CSM descriptions of the processor model. Note that communication between the processor, timer and user task is assumed to be instantaneous.

Figure 3.12: User, processor and timer task relationships.

When a task requires the services of its processor, it sends a REQUEST(t) message to its processor, indicating that it requires t units of processing time, and waits until it receives a DONE message before continuing. The processor sends ON(t) to the timer to indicate that a task has requested t time units of processing and awaits an ACK. The timer will set a timer for the current task; the duration of the timer setting is the minimum of the amount of time requested by the task and the processor's timeslicing quantum. Once the timer elapses, the amount of processing time which has elapsed is computed and the timer informs the processor of this duration by sending a STOP(t) message. If the processor receives a higher priority request while busy and is in preempt/resume mode then an OFF message is sent to the timer causing the current task to be interrupted. Figure 3.13 shows a state machine segment which could be used by a user task to request processing within the framework presented in this section.

Figure 3.13: CSM segment to request processing.

Figure 3.14 shows a possible messaging scenario related to a single processing request. The user task sends a REQUEST to the processor and is queued. After a delay, the processor schedules the task and sends an ON(t) message to the timer to start timing the task. Before the task's quantum has elapsed, the processor receives a higher priority processing request (not shown) and sends an OFF message to the timer to interrupt the current task. After the processing of the higher priority task is finished (not shown), the processor reschedules the user task and it gets the remainder of its required processing time in a single quantum (or fraction of a quantum). The timer elapses causing the timer to again send a STOP(t) message to the processor. Now the processor sees that the task's processing request has been satisfied and sends a DONE message to the task.

Figure 3.14: Possible messaging scenario for a single processing request.

Figure 3.15 shows a communicating state machine description of a processor and Figure 3.16 shows a communicating state machine description of the processor's associated timer task. In general, the priority scheme used in the processor model will execute tasks in order of highest priority to lowest and will use round-robin scheduling within priority levels if a timeslicing quantum is defined for the processor.

Figure 3.15: CSM description of a processor task.

Figure 3.16: CSM description of the timer task.

It should be stressed again that processors in MTS are not implemented as state machines which are interpreted by the simulator. This was done originally but was found to impact performance significantly. The processor model is now implemented directly as part of the simulation engine. The above state machine descriptions are an accurate representation of the functioning of the processors.

There are some notations used in the CSM description of the Processor which should be explained. The function call, *queue(request)*, results in the processing request being placed in the processor's priority queue. The function call, *select_next()*, returns the processing request from the front of the processor's priority queue; if there are no requests pending on the priority queue then this function will return nil. *C* is a local variable used to maintain information concerning the task which is currently being served by the processor; in conjunction with this, *time(C)* represents the amount of processing time still required by *C* and *prio(C)* is the priority of the task. *X* is a local variable used to maintain information about processing requests received by the processor (similar to *C*). The boolean condition, *idle?*, is used to determine if the processor is currently not serving a task - this is equivalent

to the question, "is C nil?". The boolean condition, *busy?*, is used to determine if the processor is currently serving a task - this is equivalent to the question, "is C not nil?".

There are some notations used in the CSM description of the Timer which should be explained. The global variable, *now*, records the current simulated time. The local variable, *Q_START*, is used to record the time at which the current quantum timeslice began. The boolean condition, *timeout?*, is used to determine if a receive state's activity resulted in a timeout period elapsing.

In order to allow for flexible processor allocation a processor pool abstraction has been developed. This allows the model designer to experiment with various strategies for distributing work among one or several finite sets of processors. A processor pool is composed of a *processor set*, a set of *processor characteristics* and a *processor selection rule*. The processor set identifies the processors which belong to the pool. The processor characteristics define the scheduling characteristics used by all members of the processor set. The processor selection rule defines the strategy to be used when assigning workload to elements of the pool. Currently, MTS supports three different selection strategies: least-recently-used (least recently allocated), round-robin or random.

Note that a task is allocated to a processor for its lifetime in the current implementation of MTS. Another possible implementation could have tasks reallocated every time they request processing. The changes required to realize this alternative in the MTS as it stands are quite simple. Whereas currently, a task requesting processing simply rendezvous with its processor, the task would, under the proposed strategy, first have itself reallocated using the selection rule in use for the task's pool and then rendezvous with the indicated processor.

3.4.2 Communication Network

In order to allow a model designer to introduce communication delays or unreliable communication into a model, MTS includes support for modeling a homogeneous communication network. It is assumed that tasks can always communicate directly and that the same type of communication hardware is used between all pairs of processors. There are four parameters associated with the communication network: 1) the concurrent transmission mode flag, 2) the message order flag, 3) the channel delay function and 4) the message delivery function. The concurrent transmission mode flag indicates whether or not the communication network should act like a point-to-point network (allowing multiple messages concurrent entry to the communications network) or a bus (serializing all message transmissions). The message order flag, used when concurrent transmissions are allowed, indicates whether or not messages between a processor pair can be allowed to get out of sequence. For example, assume node_i sends a message, m_1 , to node_j and then node_i sends another message, m_2 , to node_j. If message order is to be maintained, then m_1 is guaranteed to arrive at node_j before m_2 . The channel delay function is a user defined delay function which returns, for each message entering the network, a floating-point value representing the amount of time required to transmit the associated message (excluding network queuing time). The message success function is a user defined boolean function which returns, for each message sent through the network, a flag indicating whether or not the message will be lost by the network.

Note that tasks which are allocated to run on the same processor are assumed to communicate without delay. In general, two tasks on the same processor may be able to communicate by simply sharing local data space. This implies that the communication time is reduced to context switching time. Since context switching time is ignored in the

processor model, it has also been ignored here. Another important point concerning the communication network is that creating a task with no associated processor bypasses the communication network, as far as that task is concerned. Any messages sent to or from the task will be delivered immediately. Recall that tasks in MTS generally represent a software process or an independent hardware component in the system being studied. In simulation, however, it is sometimes useful to create a *pseudotask* which is a task that serves some purpose in the model but is external to the system being studied. An example of this would be a task which generates input requests for a system and, therefore, represents the users of a system rather than part of the system. A task of this nature would probably not have an associated processor in an MTS model.

This section describes the functionality of the MTS communication network in terms of CSMs. It must be stressed, however, that, as in the case of the MTS processor model, the communication network has been embedded in the simulation engine for reasons of simulator performance. This section should then be viewed as purely descriptive. For the discussion in this section, assume that a set of communicating user tasks are configured as shown in Figure 3.17; each task knows the identifier of the communication network manager. Within this context, a user task would deliver messages to the communication manager which would, in turn, forward them to the intended recipient based on the communication parameters in use. Naturally, this would lead to clumsy models if the user was forced to explicitly deal with the communication manager; this is another reason for embedding the communication network in the simulator - it has been made transparent to the model designer.

Figure 3.17: Relationship of user tasks to the communication network.

Figures 3.18 and 3.19 show a description of the communication manager and its associated timer in terms of CSMs. All messages sent by tasks are examined by the communication manager. If they are intraprocessor messages, they are sent immediately to the destination task. If they are interprocessor messages, then a delay value is computed and, based on the current network parameters, the message is queued for later delivery. The timer is a delta timer, suspending itself from the current time until the time of the next delivery and then informing the communication manager that it is time to deliver a message. The communication manager will then dequeue the message at the head of its delivery queue and send it to the intended recipient.

There are some notations used in the CSM description of the Communication Network Manager which should be explained. The global variable, *now*, records the current simulated time. The function, *delay()*, is the communication network delay function as described above. The message format, *msg(to,from,cont)*, is used by all tasks sending messages to the communication network manager; the separate parameters can be accessed as *msg(to)*, *msg(from)* and *msg(cont)* to obtain the recipient identifier, the sender identifier and the message contents respectively. The function call, *proc(identifier)*, is used to

specify the processor of the task determined by the identifier; if this is nil then the task does not use a processor and bypasses the communication network. The boolean condition, *concurrent trans?*, is a test of the concurrent transmission mode flag, described above. The boolean condition, *msg order?*, is a test of the message order flag, described above. *FT* is short for "Free Time" and is a variable which contains the time at which the communication network will next be free to start a transmission; this is used only when all message deliveries are being serialized across a single transmission medium (such as a bus). *FC* is short for "Free Channel" and is a matrix which contains all the times at which each separate channel in the communication network will next be free to start a transmission. Clearly, the usage of this is exactly the same as for *FT*, except that now since there are $n(n-1)$ transmission media a single value will no longer serve the purpose.

Figure 3.18: CSM description of the communication network manager.

Figure 3.19: CSM description of the communication timer.

There are some notations used in the CSM description of the Communication Timer which should be explained. The global variable, *now*, records the current simulated time. The boolean condition, *timeout?*, is used to determine if a receive state's activity resulted in a timeout period elapsing.

3.5 Collecting Statistic Observations in MTS

MTS provides support for collecting population or time-function statistics. In general, population statistics are used to collect information concerning temporary entities in a model (such as the amount of time a datum takes to pass through a pipeline) while time-function

statistics are used to collect information concerning permanent entities (such as processor response time statistics). Routines are also provided to generate values from various statistical distributions using distinct streams of pseudo-random numbers.

The following routines are provided for collecting statistics in MTS: *population_update(pstring, value)* and *time_function_update(pstring, value)*. In each case, *pstring* is a pointer to a C string which is used to identify the population sample and *value* is an observation from the population. *population_update()* is used to collect population statistics. The first time it is called with a given string, a file will be opened with the name "MtsStat.<compressed string>" (where <compressed string> is *pstring* with all non-alphanumeric characters removed) and *pstring* and *value* will be written to the file. Subsequent calls will result in *value* being written to the already opened file. Thus, population observations are gathered into files for postprocessing. A unix utility called *mtspost* has been developed to postprocess the population files generated by MTS. *time_function_update()* is used to collect time-function statistics. The first time it is called with a given string, a table entry will be allocated for this statistic and the statistic will be updated with *value*. Subsequent calls will result in the already started statistic being updated. The format is used for compatibility with the population statistics routine.

Generally, there are two classes of population statistics which will be of interest for models developed using MTS: 1) statistics which relate to the activities of a particular task class and 2) statistics which relate to the activities of several tasks. An example of the first type is the average amount of time spent by a certain class waiting to receive a specific type of message. An example of the second type is the amount of time required for an entity to pass through a system (set of tasks).

In the first of these two types of statistic, the amount of time can be characterized in terms

of the amount of time spent in one or more states by the class in question. Since the passage of time is accounted for solely in states, it is possible to characterize the activities of a class by collecting "time-in-state" statistics for certain of its states. This is done by introducing a flag for each state in the class definition indicating whether or not statistics are to be collected for the given state and a timestamp in the task control block (tcb) of each task indicating the time at which the state was entered. With this framework in place, it is possible to decide, for each class, which states are of interest at model definition time. When a (<class>, <state>) pair is specified for the collection of time-in-state statistics in the definition of a model, `population_update()` is used with a string of the form, "<uppercase class name>-TIS<compressed state name>", to indicate that the statistic being collected is a time-in-state (TIS) statistic. The statistic data is collected using `population_update()` and is, therefore, written to a file as described above.

The second set of statistics require design considerations during the state machine definition phase of model building. However, using the facilities provided by `population_update()`, it is possible for a model designer to specify that statistics are to be gathered at state machine definition time without worrying about the details of collection.

These statistic collection routines were designed to allow a model designer to isolate, as much as possible, the decisions concerning statistics gathering from the other logical components of model building. By using C strings to identify the populations, the designer can simply assign a name which is to be logically associated with a sample. This allows a designer to specify the populations which will be of interest for instances of a given class when the class is being defined. When defining a model using that class, the statistics of interest for that class will have already been defined. Furthermore, since the population statistics are written to files for postprocessing, the model designer does not have to be concerned with specifying histogram parameters at model definition time either.

The model will simply collect all of the data and the user can experiment with histogram parameters after the data is collected. Note that task specific statistics can be gathered by generating an unique string using task names at run time.

It is sometimes desirable to collect observations into *blocks* of data to allow estimation of confidence intervals for the population or time-function mean and to allow estimation of serial correlation of the observations. Two types of observation blocking can be used in MTS. The types of blocking provided are *fixed size* or *fixed time interval* blocking. Every time a statistic population is blocked, the estimate of the mean for the current block of observations is used to estimate the mean and variance of the population or time-function being sampled. Using this information confidence intervals for the population mean may be computed.

3.6 Summary

This chapter has presented the major design issues and concepts used in the development of MTS and its model definition environment.

First the modifications to the basic CSM model which are required to support efficient simulation were presented followed by a brief overview of the state machine interpretation process used by the run-time simulation engine.

Next, task/alias diagrams, the graphical notation used to define the interrelationships which exist between a set of tasks, were presented. This notation, along with the state diagram notation for describing tasks, provides the framework for describing MTS models. To control the complexity of state and task/alias diagrams a hierarchical abstraction mechanism, based on clusters and shadows, is used in MTS. This mechanism was also

presented. Then some basics concerning Smalltalk-80 user interfaces were presented - this information may be helpful when reading Chapters 3 and 4 which describe the MTS model development interface.

Then the functionality of the processor, processor pool and communication network abstractions which have been built into MTS were presented using state and task/alias diagrams. Finally, some of the design issues related to the definition of statistics collection were also presented.

This completes the description of the MTS run-time environment and the graphical notations which have been used in the creation of a model definition interface for the system. Chapters 4 and 5 present the user interface which has been developed for MTS.

Chapter 4

Class Behaviour Definition in MTS

The Behaviour Definition Component of an MTS class is the portion of the class definition which describes how instances of the class will function. The behaviour definition is described in terms of a CSM model. The MTS user interface supports direct manipulation editing of the structure of the state diagrams used to define task classes. The main editor for creating or modifying class behaviour definitions is a Smalltalk-80 object of class *MtsStateMachineEditor*. Transitions and each type of state also have a special editor which is used for defining the labels and parameters required for each object. These editors will be presented in this chapter.

Figure 4.1 shows the main state machine editor viewing level 0 (the top level) of a state machine from class *DPCProducer* (a subclass of *MtsStateMachine*). There are four subsections to the editor: the scrollable graphic view of the state machine, the text view of the class' instance variables, a set of editing mode soft buttons and a text view of a machine summary giving the number of states and transitions in the definition. The graphical view of the editor shows three states and five transitions. The states and transitions each have an associated label and the states have a type label: two of the states are receive states while one is a cluster state. The machine summary window shows that the state machine contains a total of four states and six transitions: these counts do not include shadows or clusters.

Figure 4.1: An instance of MtsStateMachineEditor viewing DPCProducer.

Figure 4.2: The remainder of the DPCProducer state diagram.

The portion of the state machine which is represented by the cluster state is shown in Figure 4.2. The shadow states and shadow arcs are displayed using a grey line rather than the black line used for regular arcs and states. Notice that the labels on the state machine editor windows are of the form, `<class>[/<cluster name 1>[/<cluster name 2> ...]] (level n)`, and specify an unique path name from level 0 to the portion of the state machine being viewed.

The class definition shows that instances of this class have three variables: NEXT, CONSUMER and PARMS. These will be mapped into floating-point variables stored in the task control block of the class instances when the simulation source is generated. The variable names are uppercase letters so that the C preprocessor can be used to transform the variable names into the appropriate references. The state machine editor does not do error checking to ensure that instance variables referenced in the definition of a task class are entered in the task variables view. This suggests that the task variables view may just be causing the designer to do unnecessary typing but this is not the case. In order to avoid the task of parsing the user-defined functions associated with a task class and extracting implicitly defined task variables, the interface was designed such that all instance variables would be explicitly specified by the user. Undefined instance variables will be flagged as errors either when the simulation engine source code is being generated (see Chapter 5) or when the source is being compiled, depending on the usage of the instance variable in question.

As shown above, there is a set of soft buttons on the state machine editor which are used for selecting an editing mode. Some of these modes are applicable to operations on either states or arcs. When one of these modes is selected, the buttons marked "state(s)" and "arc(s)" are used to determine which type of operation will be performed. The selected operation is carried out by simple interactive operations using the mouse as the main input

device.

The main state machine editor provides support for creating and modifying the structure of a class definition. States and arcs can be added, deleted and moved within the graphic view of the state diagram. Optional paths can be specified for the arcs representing state transitions by adding points through which a "rubber-banded line" will be drawn if the arc's originating state is not also its terminating state and through which a spline will be drawn if the arc is a self-loop on some state. In Figure 4.1, the arc from the cluster state labelled "produce an item" to the receive state labelled "?TOKEN(startProd,finCons)" is "rubber-banded" through one point while the self-loop on the receive state labelled "{REQUEST | TOKEN}" uses a spline through the optional points. Different drawing techniques are used in the two cases for aesthetic reasons.

A set of states may be clustered to create a new level in the hierarchical structure of the state machine. The result of a cluster operation is that the user is presented with a less detailed and (hopefully) more intuitively useful view of the state machine. The user specifies a rectangle on the screen which contains the states to be moved down a level in the hierarchy. These states will be replaced in the current level of the state machine by a single cluster state. To reverse the clustering operation, the user simply indicates that the cluster should be expanded by selecting "expand a cluster" mode and clicking the red button on the desired cluster. The cluster state will be replaced by the set of states it represents in the next lower level of the state diagram's hierarchical structure.

Figure 4.3 depicts some simple editing operations and the order in which the required information would be specified. The objects drawn with solid lines represent graph elements which existed prior to the editing operation while those drawn with a dotted line represent objects created or modified during the operation. Figure 4.3(a) shows a

transition being added to the state diagram. The originating and terminating states for the transition are first specified by clicking the red button on each state and then optional path points are entered by clicking the red button. Input of optional path points is terminated by clicking either the yellow or blue button. Figure 4.3(b) shows a set of states being selected for clustering. A rectangle is specified by first specifying the upper left corner and then the lower right corner using the red button. Any state that has its center inside the resulting rectangle will be moved down one level in the hierarchical structure of the state diagram. The states will be represented by a single cluster in the current level of the graph. Figure 4.3(c) shows an arc point being repositioned using the "move arc point" editing mode. The arc point to be moved is specified by pressing the red button while the cursor is positioned on the point. Then, the line segments ending at that point will be redrawn as the cursor moves until the red button is released. The cursor point, when the red button is released, will be the new position of the arc point in question.

Figure 4.3: Some simple editing operations.

Arcs may not be added incident on either shadow states or cluster states. Since a cluster state represents an arbitrary number of states, adding an arc incident on a cluster could reasonably be expected to mean that the arc is incident on all, one or some of the represented states. In the communicating state machine model, however, an arc is restricted to connect exactly two states. Therefore, the actual state to which the arc is connected would still have to be specified. Similarly, the operation of adding an arc incident on a shadow state can be viewed as adding an outgoing arc to a shadow state at a higher level of the state machine and is disallowed. In order to add arcs "between levels", the cluster state should be expanded. Then the arc can be added and the set of states can be clustered again.

4.1 Editing State and Transition Parameters

Using the "set arc/state parms" mode, an arc or state may be specified using the red button and a new editor will be created which will enable the user to set parameters which are relevant to the specified object. These editors will be presented in detail in the following sections.

4.1.1 Transition Parameter Editing

Figure 4.4 shows an example of an editor which may be used to specify a label for an arc, the actions which should be performed when the transition represented by the arc occurs and the conditions under which that transition should occur.

MTS Transition Editor

label:	?TOKEN(startProd,flnCons) \ NEXT!TOKEN(startProd,flnCons)					
create	class:					
action:	assignment function will be performed					
kill	task:					
send	to:	NEXT	entry:	TOKEN	type:	PARMS
assign function text:						
<pre> int #name#(tp) struct tob *tp; /* task pointer */ { PARMS = tp->mtype; } </pre>						
select function:						
<pre> int #name#(tp) struct tob *tp; /* task pointer */ { return(tp->mentry == TOKEN); } </pre>						

Figure 4.4: MTS transition parameter editor.

The top line of the editor is a text editor which is used to modify the label which will be displayed on the arc in the graphic view of the state machine. Notice the backslash included in the label; all backslashes will be replaced by carriage returns when the label is displayed.

The next four lines of the editor are divided into two sections. The leftmost portion of each line contains a soft button which is "on" if the associated action is to be performed as part of transition and "off" otherwise. The rest of the line is used to provide information concerning the action. Notice that the soft buttons for the four actions are presented in the order in which the actions will be performed during transition; this is to aid the user in determining the effect of transition.

For the *create* operation, the only information required is the class of the task to be created. For the *assign* operation, a C language template for the function is presented. It is the user's responsibility to write a code segment which will accomplish the desired task. Notice that the name shown for both the assignment function and the arc selection function is "#name#". This is a function name template which will be replaced by the interface when the simulation engine source code is generated for the model. To define the *send* operation a receiver, a receiver entry and the message type must be specified. The receiver is specified using an instance variable of the sender which contains the task identifier of the intended recipient. The entry should be valid for the type of task which has been specified as the message receiver. The message type is used to specify an optional parameter for the message. This can be either a scalar value or it can be the name of one of the active task's instance variables. Using the library of functions for manipulating collections of data and a task instance variable, arbitrary length parameter lists may be passed with messages. The only information required for the *kill* operation is the identity of the task to be killed. This can be specified by entering the name of the active task's instance variable which will contain the identifier of the task to be killed when the kill operation is performed. Typically, the transition's assignment function will be used to initialize the variable to the proper task identifier.

4.1.2 Cluster State Parameter Editing

Figure 4.5 shows an example of a cluster state parameter editor. This is the simplest type of editor since a cluster state has the fewest parameters. The only parameters which may be modified for a cluster state are the state label and the positioning of the label on the graphic view of the state machine.

The label is modified as was already described for a transition's label. The positioning of the label is handled differently for a state. There are two rows of soft buttons in the editor. One row determines the horizontal positioning of the label with respect to the state while the other row determines the vertical positioning of the label with respect to the state. According to the setting of the soft buttons in Figure 4.5, the label "produce an item" would be positioned to the left and vertically centred with respect to the associated cluster state. The positioning of the label with respect to the state is shown in Figure 4.1.

MTS Cluster State Editor		
label:	produce an item	
left	horizontally centered	right
above	vertically centered	below

Figure 4.5: MTS cluster state parameter editor.

4.1.3 Connect State Parameter Editing

Figure 4.6 shows an example of a connect state editor. In addition to the portions of the editor which have already been described, there is a section for specifying the order in which the arc selection functions of outgoing transitions will be tested and a soft button for specifying whether or not the associated state is the initial state of its state machine.

In some cases, the order in which the arc selection functions are executed may affect the interpretation of the state machine. The list in the left portion of the editor shows the current order in which the functions will be executed. The two soft buttons in the right portion of the editor, "rotate list" and "exchange elements", may be used to reorder the list. Clicking the red button on the "rotate list" soft button causes the last element of the list to move to the first position, shuffling all other elements down one position. Clicking the red button on the "exchange elements" soft button causes the system to prompt for the list

indices of two elements which will be swapped.

MTS Connect State Editor		
label:	can I start a production?	initial state
left	horizontally centered	right
above	vertically centered	below
Outgoing arc testing order:		
1) startProd-finCons<N/...startProd+1,finCons> 2) startProd-finCons>=N/...startProd,finCons)		rotate list
		exchange elements

Figure 4.6: MTS connect state parameter editor.

4.1.4 Delay and Process State Parameter Editing

Figure 4.7 shows an example delay state parameter editor. The only new part of the editor is the text view of the C source code for the state's delay function. Again the function name template, "#name#", will be replaced by the interface when the simulation source is generated. Process state editors are the same except that the editor includes the process delay function text.

MTS Delay State Editor

label:	an example delay state		initial state
left	horizontally centered	right	
above	vertically centered	below	
Outgoing arc testing order:			
1) outgoing arc 1 2) outgoing arc 2			rotate list
			exchange elements
delay function:			
<pre>double #name#(tp) struct tob *tp; /* task pointer */ { /* return a value from a uniform distribution over the range 0..100 */ extern double uniform(); double delay; delay = uniform(0.0, 100.0, 1); return(delay); }</pre>			

Figure 4.7: MTS delay state parameter editor.

4.1.5 Receive State Parameter Editing

Figure 4.8 shows an example receive state parameter editor. Apart from what has already been described, there is only one new section for this editor. That is the portion of the editor which is used to specify the message reception parameters for the state. The parameters which may be specified are the message sender, the timeout period and the entries which will be accepted. If no sender is specified, then the task will accept messages from any other task. If a timeout is to be used, then the timeout soft button must be turned "on" and a duration value specified. A list of acceptable entries for the current state must be

specified. For example, in Figure 4.8 the task will accept messages at entries REQUEST or TOKEN.

MTS Receive State Editor				
label:	?(REQUEST TOKEN)\			initial state
left		horizontally centered	right	
above		vertically centered	below	
Outgoing arc testing order:				
1) ?REQUEST/. 2) ?TOKEN(startProd,fi...startProd,finCons)				rotate list
				exchange elements
msg from:	?	time out	duration:	0.0
entries:	REQUEST TOKEN			

Figure 4.8: MTS receive state parameter editor.

4.2 State Machine Yellow Button Editing Operations

Figure 4.9 shows the yellow button menu provided for the graphical view of the state machine. The first four menu items provide means of altering or supplementing the user's current view of the state machine while the last two provide services for saving or clearing the state machine.

top level
up a level
down a level
open on lower level
save machine
clear machine

Figure 4.9: MTS state machine graphical view yellow button menu.

The menu items and an explanation of each one are given below:

- 1) *top level.* This menu item causes the graphic view to display level 0 of the currently displayed state machine.
- 2) *up a level.* This menu item causes the level above the one currently displayed in the hierarchical structure of the state machine to be displayed.
- 3) *down a level.* This menu item waits for the user to specify a cluster state using the red button and then displays the level of the state machine represented by the specified cluster state.
- 4) *open on a lower level.* This menu item waits for the user to specify a cluster state using the red button and then opens a new editor viewing the level of the state machine represented by the specified cluster.
- 5) *save machine.* This menu item prompts the user for a name and then saves the state machine definition as a subclass of *MtsStateMachine* in the Smalltalk-80 class hierarchy. Note that if this is the first state machine saved in the Smalltalk-80 image being used, the class category view of the system browser may need to be updated before the state machine class definition can be viewed. Once the class category view is updated, the state machine class definition will be in category "MTS-Machines".
- 6) *clear machine.* This menu item reinitializes the state machine being viewed to an empty state machine.

4.3 Summary

This chapter presented the editing tools used for the creation and modification of MTS task class definitions. Without describing in detail all of the editing operations provided by the tools, the kinds of operations and a representative set of the operations supported were shown. Appendix B presents a complete description of the editing modes and operations

supported by the main state machine editor.

Each element of an MTS state diagram has a default representation which is used when the object is added to the diagram. For instance, a transition is represented by a straight line with its label centred on the line's midpoint. Using direct mouse manipulations, the graphical representation of the elements of the diagram may be modified to clarify the definition.

Each type of object in a state diagram (state or transition) has an associated editor which is used to specify parameters for the elements of a state diagram. These editors were presented in this chapter.

Using the editing modes of the main editor, the menus and the parameter modification editors, state diagrams defining the behaviour of task classes for MTS models may be created, modified and saved for future reference. As a result, it is a simple matter to create and extend a library of MTS task classes.

Chapter 5

Model Definition in MTS

As described in Chapter 3, an MTS model is defined by specifying a set of static tasks such that each element of the set is an instance of a task class which is defined for the model. The behaviour of MTS task classes are defined in terms of CSMs as described in Chapter 4. This chapter presents the user interface developed for the definition of simulation models using a set of task classes as input.

The graphical notation used in MTS to describe the static task set and the interrelationships which exist between elements of that set is called a static task/alias diagram. In general, a task represents either a software process or an independent hardware component of a system. An alias represents knowledge held by one task concerning the identity of another. In the MTS user interface, the main editor for creating or modifying models is a Smalltalk-80 object of class *MtsModelEditor*. This editor supports direct manipulation of a model's static task/alias diagram and allows the user to access the other editors required to completely specify a model.

In MTS, a model is defined in the following order:

- 1) The classes which are to be used in the model must be specified.
- 2) The static task/alias diagram must be created.
- 3) The dynamic configuration information for each class, from which dynamic tasks are to be created, must be defined. This specifies the relationship of dynamically created instances of the task to the members of the static task set and, as a result,

must be specified after the static task set is created.

- 4) The processor pools, communication parameters and run-time parameters must be specified to complete the model definition.

The reason for the order given above is quite simple. The static configuration diagram requires information concerning the different classes such as instance variable names. The dynamic configuration diagrams require both class information and information concerning the static task set, such as task names. Therefore, although the dynamic configuration information is considered to be part of the class definition, it is a model specific component of the class definition and must be specified after the static task set is defined.

The main model definition editor, presented in this chapter, and the main state machine editor, presented in Chapter 4, have several common features. As described in Chapter 2, MTS uses clustering and shadowing to support hierarchically structured graphs with both the static task/alias and state diagrams. Also the main editing modes provided by the two editors and the associated operations are very similar.

Figure 5.1 shows the main editing window viewing level 0 of a model from class DPC (a subclass of MtsModel). There are five subsections to the editor: the scrollable graphic view of the static task/alias diagram; the text view of the classes in use in the model (top left); the text view showing text to be included in the generated simulation source code (top right); a set of editing mode soft buttons; a text view of a model summary giving the number of tasks, aliases, processors, pools and classes in the definition.

Figure 5.1: The main MTS model editor viewing the definition of DPC.

The class list window in the upper left corner of the editor shows the classes which are in use in the model using the format "<classname> - <Smalltalk classname>". The <classname> is the name used to identify the class throughout the model (for instance, when a new instance of the class is to be created) while the <Smalltalk classname> identifies the Smalltalk-80 class which contains the definition of this task class. Notice that the task/alias diagram shows both the name and class (type) of the tasks. The class list view of the editor has a yellow button menu which provides operations for adding, deleting and changing class mappings and for opening an editor to modify the dynamic configuration information of a class. These functions will be presented in the following sections.

The text editing window in the upper right hand corner of the editor can be used for commenting models and/or for including "#include" references or global data to be replicated in the simulation engine source code when it is generated. This provides an easy method for including data which will be shared by a set of tasks and for including routines to manage access to the shared data, if necessary.

Notice, in Figure 5.1, that the task/alias diagram has an arc from *producer5* to *consumer* labelled "CONSUMER, NEXT". This indicates that *producer5* knows *consumer* by two different aliases: CONSUMER and NEXT. In general, when a task, t_1 , has a set of aliases for another task, t_2 , all of t_1 's aliases for t_2 will be represented by a single, appropriately labelled, directed arc. There is a yellow button menu provided in the graphic task/alias view of the editor which provides various editing functions. This menu will be presented in a later section.

The model summary shows the number of classes, tasks, processors, pools and aliases

which have been defined in the model. As in the state machine editor, these totals do not include shadows or clusters for the tasks and aliases.

As shown in Figure 5.1, there is a set of soft buttons on the static task/alias editor which are used for selecting an editing mode. Some of these modes are applicable to either task operations or alias operations. For these modes, the state of the "task(s)" and "alias(es)" buttons are used to determine which type of operation is to be performed. As with the state machine editors presented in Chapter 4, the editing operations involve simple user interactions using the mouse as the main input device. The static task/alias diagram editor provides support for creating and modifying the graph which represents the static component of an MTS model. Tasks and aliases may be added, deleted and repositioned within the graphic view of the diagram. Optional paths may be specified for the directed arcs representing aliases in the diagram by adding points through which a "rubber-banded line" will be drawn. A set of tasks may be clustered to create a new level in the hierarchical structure of the task/alias diagram. Generally, this function is used to group components of a model together which are logically related in some way. The operations for modifying the structure of a static task/alias diagram are very similar to the operations presented in Chapter 4 for editing the structure of a state diagram and need not be presented in detail.

To allow the user to specify unique names and initialize instance variables for the tasks in a static task/alias diagram, a task parameter editor, as shown in Figure 5.2, is used. To access this editor, the user simply selects the "set task parms" mode and uses the red button to specify the task to be edited.

input task	
name:	Input task
modify var lists below - accept changes:	
LOW ← 0.0 . CREATION ← . TIMESTREAM ← 1.0 . PRODSTREAM ← 2.0 . PRODUCER ← . HIGH ← 25.0 .	
PRODUCERS →producer1(producer) →producer2(producer) →producer3(producer) →producer4(producer) →producer5(producer)	

Figure 5.2: MTS task parameter editor.

The top portion of the task parameter editor is used to edit the name of the task. The remaining part of the editor is divided into two sections which are used to display the contents of the task's instance variables. One section of the editor is used to display and edit task variables which contain numeric information while the other section is used to display variables which contain aliases. The numeric values are specified and displayed as floating-point numbers. The alias specifications are in the format, "→<task name>(<task type>)", and indicate that this variable will contain the identifier of the named task when the simulation source code is generated.

The information associated with each instance variable is specified as a list terminated by a period. If the list contains only a single value, then the instance variable specified will contain that value when the simulation is run. If, however, the list contains more than a single value then the task's instance variable will be initialized to be the index of a collection header for a collection containing the specified values.

5.1 Static Task/Alias Yellow Button Menu

Figure 5.3 shows the yellow button menu provided in the static task/alias view of the model editor. The first four menu items provide operations for changing the current view of the displayed model. These operate exactly as in the yellow button menu of the graphic view in the main state machine editor and will not be described here (see section 4.2). The next three menu items provide operations to allow the user to view various components of the model. The last three menu items provide operations for storing the model for later use, clearing the model or generating C source code to implement the simulation engine for the model.

top level
up a level
down a level
open on lower level
view processor pools
view communication parms
view run time parms
save model
clear model
write simulation engine source

Figure 5.3: Static task/alias yellow button menu.

The operations provided by the menu are described below:

- 1) *view processor pools*. This menu item is used to open an editor which will allow the user to view or modify the model's processor pools. This editor will be presented in a later section.
- 2) *view communication parameters*. This menu item is used to open an editor which will allow the user to view or modify the parameters for the model's communication network. This editor will be presented in a later section.
- 3) *view run time parameters*. This menu item is used to open an editor which will

allow the user to view or modify the model's run-time parameters. This editor will be presented in a later section.

- 4) *save machine*. This menu item prompts the user for a name and then saves the model definition as a subclass of MtsModel in the Smalltalk-80 class hierarchy. Note that if this is the first model saved in the Smalltalk-80 image being used, the class category view of the system browser may need to be updated before the model class definition can be viewed. Once the class category view is updated, the model class definition will be in category "MTS-Models".
- 5) *clear machine*. This menu item reinitializes the model being viewed to an empty model.
- 6) *write simulation engine source*. This menu item will cause the system to prompt the user for a file name. When the user has entered a name, the source code which will implement the model will be written to the specified file. Once written, the source must be compiled and linked with the rest of the simulator and then run.

5.2 Class List Yellow Button Menu

Figure 5.4 shows the class list yellow button menu. The first three menu items are used to modify the class behavioural definition for some class in the model while the last is used to modify the dynamic configuration information for some class in the model. Some of the operations on this menu affect only the currently selected item in the class list view. The currently selected item (if an item is selected) is displayed using white lettering on black background. List items may be selected or deselected by positioning the cursor on the item and clicking the red button.

Figure 5.4: Model editor class list yellow button menu.

The operations provided by the menu are described below:

- 1) *add class*. This menu item is used when a new task class is to be added to the model's class set. The class is specified by entering the name to be used for the class within the model and then specifying the Smalltalk-80 class (a subclass of *MtsStateMachine*) which defines the structure of instances of the class.
- 2) *delete class*. The system will delete the currently selected class in the class list view. If there is no class selected when this menu item is invoked the request will be ignored.
- 3) *change class mapping*. The system will prompt the user for the information required to change the class mapping of the currently selected class in the class list view. The user will be asked to specify a new Smalltalk-80 class to define the selected class (as in the "add class" operation, above). Validity checking will be performed by the system to ensure that there are no inconsistencies introduced to the model by this operation (such as task aliases which use a variable which is no longer defined for instances of the class).
- 4) *edit dynamic configuration*. The system will open and display a new editor viewing the dynamic configuration information of the currently selected class in the class list view.

5.3 Editing Dynamic Task/Alias Information

To allow a user to specify optional information concerning the initial state of dynamically created instances of a class, a dynamic configuration editor, as shown in Figure 5.5, is used. This editor is opened using the class list yellow button menu, as described in section 5.2.

Figure 5.5: MTS task class dynamic configuration editor.

The top line of the dynamic configuration editor is used to specify a name template for the dynamically created instances of the class. The number of dynamically created instances of a class is always known and all occurrences of "#" in the name template will be replaced by the string representing this number to derive a task name. This provides a simple means of supplying unique names for the dynamically created instances of a class.

The portion of the editor below the name template contains the dynamic task/alias diagram for the class. Notice that the diagram contains aliases only from the class prototype to tasks which are assumed to belong to the static task set of the model. There is a set of editing mode soft buttons to support adding, deleting and moving the tasks and aliases in the diagram.

The portion of the editor below the dynamic task/alias diagram is used to specify the pool allocation information to be used with dynamically created instances of a class. The pool allocation can either specify the name of a statically defined processor pool in the model or it can specify "creator". A pool specification of "creator" indicates that the task should be allocated to the pool used by the dynamic task's creator. The task will be allocated to a processor in a specified pool according to that pool's selection rule. Similarly, the priority can specify a value for the dynamic task's priority or it can specify that the priority of the task's creator should be used. Note that the pool allocation and instance variable initialization views of this editor specify that changes should be accepted. This means that after editing changes have been made, the user should select "accept" on the view's yellow button menu to make the changes permanent.

The last portion of the editor is used to specify initial values for the instance variables of the dynamically created task. These values are specified exactly as shown for the static tasks,

with one exception. In the dynamic task editor, a value in the variable editing window can also be specified using the format, "<task name>/<var name>", where <task name> is the name of any task in the model's static task set and <var name> is the name of an instance variable defined for that task. The result of this specification is that the value of the specified task's instance variable, *when the dynamic task is created*, will be used when initializing the dynamically created task's instance variable pool.

5.4 Editing Processor Pool Information

To specify the number, size and characteristics of processor pools in an MTS model, a processor pool editor, such as the one shown in Figure 5.6, is used. This editor is opened by selecting the "view processor pools" item on the static task/alias yellow button menu, as described in section 5.1. The processor pool editor provides mode soft button which can be used to add, delete, move or edit processor pools in the model's pool set. Figure 5.6 shows the processor pools being used in the DPC model presented earlier. In this case there are six pools, each of size 1. None of the pools have a defined timeslicing quantum and all use preempt/resume scheduling.

model processor pool

name: node1(1) characteristics: preempt/resume, quantum: undefined	
name: node2(1) characteristics: preempt/resume, quantum: undefined	name: consumerNode(1) characteristics: preempt/resume, quantum: undefined
name: node3(1) characteristics: preempt/resume, quantum: undefined	
name: node4(1) characteristics: preempt/resume, quantum: undefined	
name: node5(1) characteristics: preempt/resume, quantum: undefined	
add	delete
move	edit

Figure 5.6: MTS processor pool editor.

5.5 Editing Processor Pool Parameters

To modify the parameters of processors in a given pool, a pool parameter editor, such as the one shown in Figure 5.7, is used. This editor is opened by clicking the red button on the rectangle representing a processor pool, in the processor pools editor described in section 5.4, when the "edit" soft button is turned "on". A pool parameter editor is divided into two major sections: one section defines the characteristics of the pool or of processors in the pool and the other section is used to allocate tasks from the static task set to run on members of the pool.

The pool has a name, a size (the number of processors in the pool) and a selection rule which specifies how tasks should be allocated to elements of the pool (one of: least-recently-used, round-robin or random). The processors in the pool have a timeslicing quantum (either "undefined" or a floating-point value) and a scheduling mode (either preempt/resume or head-of-the-line).

pool editor	
name:	consumerNode
pool size:	1
quantum:	undefined
selection rule (click to change)	random
preempt/resume	head-of-the-line
----- consumer - 0 server - 0 -----	

Figure 5.7: MTS pool parameter editor.

The bottom window in the editor contains a list showing the tasks which have been allocated to run on elements of the pool and the processing priority associated with each task. A yellow button menu is provided which supports operations for allocating tasks to the pool or deallocating tasks from the pool.

5.6 Editing Communication Network Parameters

To modify the parameters of the communication network of a model, the communication parameter editor is used. This editor is opened by selecting the "view communication parms" item on the static task/alias yellow button menu, as described in section 5.1. Figure 5.8 shows the communication parameter editor.

The editor is composed of two soft buttons and two text editors. The two soft buttons indicate whether or not the communication network supports concurrent message transmission and whether or not message order is maintained if concurrent transmission is supported. The text editors are for the model's communication delay and message delivery success functions. For each of these functions, a default functions is always provided. The defaults specify that all messages are delivered successfully (shown) and that all message transmissions take zero units of simulated time (not shown). Presenting the user with default functions also provides the correct function parameters rather than forcing the user to look them up elsewhere.

model communication parameters	
concurrent transmission allowed	message order maintained
communication delay function:	
<pre>double deliveryDelay(fromIndex, toIndex, entry, mtype) int fromIndex; /* index of from processor */ int toIndex; /* index of to processor */ struct entry+rec entry; /* receiver msg entry */ double mtype; /* message type */ { extern double uniform(); double delay; delay = uniform(0.0, 10.0, 4); }</pre>	
message success function:	
<pre>int deliverySuccess(fromIndex, toIndex) int fromIndex; /* index of from processor */ int toIndex; /* index of to processor */ { /* this function could be a deterministic function based on knowledge of */ /* the message sender and receiver or it could be probabilistic (possibly */ /* using these parms). */ return(TRUE); }</pre>	

Figure 5.8: MTS communication parameter editor.

5.7 Editing Model Run-time Parameters

To modify the run-time parameters of a model, the run-time parameter editor is used. This editor is opened by selecting the "view run time parms" item on the static task/alias yellow button menu, as described in section 5.1. Figure 5.9 shows the run-time parameter editor used in MTS. The run-time parameter editor is used to specify statistics to be collected, seed values for random number streams and stopping conditions for the simulation run.

run time parameters	
collect time-in-state statistics for: <class> - <state label>	
----- input - interarrival delay server - consumption delay server - ?DATA producer - ?TOKEN(startProd,finCons) -----	
random number stream seeds - accept changes after editing	
format: <stream number>: <value>.	
1: 12345. 2: 23456. 3: 34567. 4: 45678.	
simulation stopping conditions:	
maximum simulation time:	10000
class tests: - accept changes after editing	
format: <class>/<entry> <'receive' or 'send'> <#>.	
input/REQUEST send 300.	

Figure 5.9: MTS run-time parameter editor.

The top window of the editor specifies the "time-in-state" statistics which should be collected during the run. Each entry in the list specifies that, for a given class, statistics should be collected on the amount of time spent in the state each time an instance of the class enters the state.

The second window in the run-time parameter editor is used to specify seed values for random number streams. MTS provides routines for selecting values from various statistical distributions using distinct streams of pseudo-random numbers. Each stream has an associated integer identifier. For instance, in Figure 5.9 random number stream 1 is seeded with the value 12345.

The last section of the run-time parameter editor is used for specifying stopping conditions for the simulation. One type of stopping condition simply specifies the maximum simulation time; when the simulated time reaches this value the simulation is terminated. The second type of stopping condition is a counter on class entry references. Class entries are referenced for either sending a message or for receiving a message. The last window on the editor is used for specifying maximum reference counts for <class>/<entry> pairs. For instance, Figure 5.9 shows that the simulation should stop when tasks of class "input" have sent more than 300 REQUEST messages.

5.8 Summary

This chapter has presented the editing tools used for the creation and modification of MTS models. As in Chapter 4, a brief description of the editors was given rather than a complete description of the use of these editors. Appendix B presents a complete description of the editing modes and operations supported by the task/alias diagram editor.

Similar to the elements of state diagrams presented in Chapter 4, the elements of task/alias diagrams have default representations which may be modified, using the direct manipulation editors, to clarify the model. Editors are provided to allow the initial state to be specified for individual tasks and to set model specific parameters such as communication delays and random number stream seeds.

Using these editing tools, an MTS model may be defined, modified and saved for future reference. Thus, it is a simple matter to create a library of simulation models. The simulation source code may be generated from within this environment. Appendix A shows the general structure of the generated source code files and compares a simple model defined in MTS with its generated source file.

One final point should be made concerning the use of the "accept" menu item in certain of the text editors presented in this chapter. Any text editor in Smalltalk provides an "accept" menu item which is used to indicate to the system that changes to the text in a window should be made permanent (ie. saved). In the MTS interface, the acceptance of text has been made automatic in all cases except where it has been found necessary or desirable to perform some error checking on the format of the input text. So, any text editor which requires text in a specific format (such as that used in the random number stream seed editor) will also require explicit use of the "accept" menu item.

Chapter 6

A Case Study of Distributed Mutual Exclusion

This chapter presents the results of a study done using MTS. In this section, the problem studied and a brief literature survey will be presented. The following sections will present two algorithms, in detail, which have been implemented and studied using MTS. Then the results of the simulation studies will be presented along with the asymptotic message and bit complexities of the two algorithms.

In a distributed network of processors, each node is to occasionally execute a segment of code - the *critical section* - with the condition that at most one node will ever be executing that segment at any given time. A protocol must be developed which specifies what each node will do when it wants to enter the critical section and when it exits the critical sections - these sets of actions will be called the *entry protocol* and *exit protocol*, respectively. The protocols must be distributed in the sense that each node executes the same protocols and no node acts as a central controller.

For the solution to be correct, it should exhibit certain properties called *safety* and *liveness*. Safety is the main assertion which has already been stated - that is, that no more than one node will ever be executing the critical section at the same time. Liveness can be further divided into two components: *freedom from deadlock* and *freedom from starvation*. Freedom from deadlock specifies that, assuming that every node which enters the critical section eventually exits the critical section, at least one node will always be able to enter the critical section after it has executed the entry protocol. Freedom from starvation specifies

that, assuming that every node which enters the critical section eventually exits the critical section, any node which requests access to the critical section will eventually be granted access. The solution should also be fair, allowing equal access to all processes requesting access, and should make no assumptions concerning the timing of events within the participating processes.

In [12] a system of logical clocks is presented which can be used to order the events in a distributed system. Assuming that the operations of sending and receiving messages are events in the system and that $C_i(A)$ is the logical clock value for event A in process P_i , a partial ordering of events may be established if the following conditions hold:

- 1) If A and B are events in process P_i and A precedes B , then $C_i(A) < C_i(B)$.
- 2) If A is the sending of a message by process P_i and B is the receipt of that message by process P_j , then $C_i(A) < C_j(B)$.

To establish a total ordering of events the following rule is used:

$$A \text{ precedes } B \Leftrightarrow C_i(A) < C_j(B) \vee (C_i(A) = C_j(B) \wedge P_i < P_j)$$

Thus, the process identifier is used to break ties in logical clock values. Using this total ordering and a distributed event queue, [12] presents a solution to the distributed mutual exclusion problem. A node requesting access to the critical section sends a timestamped request message to all other nodes and awaits a reply from every other node. When the requesting node has a reply, timestamped later than the requestor's request, from every other node in the network the requestor knows that the critical section is free for use. This method was later optimized in [18] by reducing the number of messages sent during the entry protocol. The method was further refined in [4] through the use of an explicit privilege token. In this algorithm, a token representing the privilege to enter the critical section is transmitted through the network. In this case, the requesting node would first

2 of /de 2

Micro-D

determine whether or not the token is currently available - if it is then no messages are required since no other node could be in the critical section.

All of these results, however, are for a complete network in which each node can communicate directly with every other node and the size of the network is known. More recently, some of these results have been extended to other network topologies. In [14] three algorithms are presented which achieve distributed mutual exclusion on a ring network. To the author's knowledge, the only algorithm which extends these results to work on an arbitrary network is presented in [9]. Author's initials will be used throughout this discussion to designate algorithms. Thus, this algorithm will be referred to as the HPR algorithm.

The HPR algorithm claims to achieve message complexities as good as those achieved by the algorithms which utilize knowledge of the network topology without using any such knowledge. This is achieved by including $O(n)$ control information in every request to enter the critical section and $O(n)$ information in the explicit token. This causes the bit complexity of the resulting algorithm to be considerably higher than the complexity of the algorithms which utilize network topology knowledge.

In the next sections, the HPR algorithm will be presented in more detail along with a new algorithm based on the algorithm presented in [18]. The results of simulation studies conducted which model the communication time for these two algorithms, using the assumption that message transmission time is proportional to the amount of information contained in a message, will be presented. The simulation results are shown for various network topologies and for a range of request rates on each network topology.

6.1 The HPR Algorithm

The HPR algorithm uses an explicit token to represent the right to enter the critical section - a node which owns the token may enter the critical section as often as it wishes. When a node which does not own the token wants access to the critical section, it sends a timestamped request through the network using a total ordering of requests as described above. Each node in the network maintains a set of the most recent requests received from every other node in the network and the token contains an array showing the time at which each node in the network last gave away the token. Therefore, the owner of the token can easily determine the earliest unsatisfied request by finding the earliest request such that the logical time on the token for the associated node is less than the logical time of the request. Routing the token through the network to the requestor informs the requestor that the critical section has been granted to it.

The algorithm makes the following assumptions: 1) communication is over bidirectional communication links and two nodes are said to be neighbours if they are directly connected by communication links, 2) nodes communicate by sending messages which are neither lost nor corrupted, 3) messages are delivered after a finite but unknown delay, 4) the order of delivery of messages being transmitted on a common communication link is not necessarily FIFO and 5) the network is connected.

To request the token, a node sends *REQUEST*((*reqOrigin*,*reqTime*),(*sender*,*alreadySeen*)) to all of its neighbours. *reqOrigin* is the node identifier of the node which originated the request. *reqTime* is the logical clock value associated with the request. *sender* is the node identifier of the last node to forward the request. *alreadySeen* is a list containing the node identifiers of all nodes which have seen or are about to see the request. Thus, *reqOrigin*

and *reqTime* are used to uniquely identify each request in the system while *sender* and *alreadySeen* are used to control the transfer of information through the network.

The token is transmitted using a message of the form, *TOKEN(elec,lud)*. *elec* is the node identifier of the node to which the token is being transmitted. *lud* is an array which shows when each node in the network last gave up ownership of the token. Thus, *elec* is used when the token is being routed through the network and *lud* is used to determine the next request to satisfy.

Let $node_i$ be any node in the network. $Node_i$ maintains the following local information:

- 1) *neighbours*. This is an array containing the set of nodes to which $node_i$ is directly connected by communication links.
- 2) *C*. This is $node_i$'s logical clock which has an initial value of zero (0).
- 3) *tokenHere*. This is a boolean value indicating whether or not $node_i$ has the token. Initially, all nodes, except one, will have *tokenHere* set to false. At any time, at most one node will have *tokenHere* set to true.
- 4) *inCS*. This is a boolean value indicating whether or not $node_i$ is using the critical section. Notice that $inCS = true \Rightarrow tokenHere = true$.
- 5) *reqArray*. This is an array of request sets in which each set is associated with a specific neighbour of $node_i$. Each set contains the requests received by $node_i$ which came directly from the associated neighbour. The elements of each set are of the form $(reqOrigin, reqTime)$ and identify requests uniquely within the network.

The complete description of the HPR algorithm can be found in the original paper. Rather than reproduce the algorithm description here, the functional description of the MTS implementation will be used to describe the details of the algorithm. This description is

given in the next section.

6.2 MTS Implementation of the HPR Algorithm

Let node_i be any node in the MTS model of a network executing the HPR algorithm.

Figure 6.1 shows a tasks/alias diagram segment showing the components of node_i .

Figure 6.1: Node Structure for the MTS Implementation of the HPR Algorithm.

The *Invoker* is the process which requests access to the critical section. The *Transmitter* is responsible for selecting the next node to receive the token, updating the token and sending it. The *Router* is the node's interface to the rest of the network - all messages from either the Invoker or the Transmitter are sent via the Router and all messages coming into the node from the network come via the Router. To simplify the implementation, the Router has also been made responsible for some processing of the incoming messages. The task

identifier of Router_i is used as the node identifier of node_i - this allows neighbouring nodes to send messages bound for node_i to the same task identifier used in their local tables. Any other approach would have resulted in an extremely awkward implementation of the algorithm.

It is important to note that the intention of this simulation was to study the messaging delays of the algorithm. Therefore, any overhead resulting from this multitasking implementation at each node is ignored. Also, communication between processes at node_i is assumed to be instantaneous.

In order to implement the algorithm using the task structure shown in Figure 6.1, some new messages have been added to the algorithm. Router_i sends a *GRANT* message to Invoker_i when the Router receives a token destined for node_i to indicate to Invoker_i that the critical section has become available. Both Router_i and Invoker_i send *TRANS* messages to Transmitter_i when the token should be forwarded to another node in the network, if there are outstanding requests.

The state diagrams presented in this section use the following notations: "==" is used for the binary test "is equal?", "!=" is used for the binary test "is not equal?", "=" is used for assignment, "&" is used for logical AND and "|" is used for logical OR.

Figure 6.2: HPRInvoker task class functional description.

Figure 6.2 shows the functional description of the Invoker task class, called *HPRInvoker*. Invoker_i simply loops indefinitely between the noncritical section and the critical section, executing the entry and exit protocols each time it does so. The Invoker first checks to see whether or not the token is already resident at the node and, if so, simply enters the critical section. If the token is not present, the task packages and sends a REQUEST message to the Router which will forward it to all of node_i 's neighbours. Invoker_i will then wait until Router_i sends a GRANT message indicating that the token is now resident at node_i . After executing the critical section, Invoker_i sends a TRANS message to Transmitter_i , indicating

that Transmitter_i may now give the token to another node if any outstanding requests are known.

Figure 6.3: HPRTransmitter task class functional description.

Figure 6.3 shows the functional description of the Transmitter task class, called *HPRTransmitter*. Transmitter_i waits for TRANS messages from either Invoker_i or Router_i. When it receives one, it attempts to select the next node which will receive the token by scanning the *lud* array in the token and node_i's *reqArray* to determine the oldest unsatisfied request. Before sending the token, the *lud* entry for node_i is updated to show the time at which node_i last gave away the token (ie. the current time). If Transmitter_i

knows of no outstanding requests, the token will remain at node_i. Recall that the task identifier of a node's Router is used as the node's identifier. All task identifiers used in MTS are non-negative integers. Therefore, if ELECTED is negative, then Transmitter_i does not know of any outstanding requests and the token will remain at node_i.

Figures 6.4 to 6.8 show the hierarchically structured functional description of the Router task class, called *HPRRouter*. The figure captions show the MTS state machine editor label associated with each portion of the diagram to provide a frame of reference.

Figure 6.4: HPRRouter (level 0)

Figure 6.4 shows the top level of the state diagram for the Router task class. Router_i waits

for either outgoing messages (the message sender was either Invoker_i or Transmitter_i , therefore the message is being sent out of the node) or incoming messages (the message sender was neither Invoker_i nor Transmitter_i). When i receives a message, it determines the direction (incoming or outgoing) and type of the message and handles it appropriately. The details associated with each type of message and each direction are shown in the following segments of the state diagram.

Figure 6.5: HPRRouter/handle incoming traffic (level 1)

Figure 6.5 shows the portion of the functional description for HPRRouter which deals with incoming messages. The message must be one of two types - either the token or a request. If it is a request, Router_i attempts to find and remove an older request from the originator of

the request just received. Since this is done every time a request is received, there can never be more than one outdated request stored in node_i 's local data structures for each node in the network. The details of handling each type of incoming message are shown in the following state diagram segments.

Figure 6.6 shows the portion of the state diagram which describes the handling of incoming requests by an instance of class HPRRouter. First it must determine whether or not the REQUEST message received is outdated - it is possible for node_i to receive a request after that request has been satisfied and after a new request from the same node has already been received if it is assumed that message delays are unknown and the message ordering over communication links is not necessarily FIFO. If node_i 's *reqArray* contains an entry from the originator of the current request such that the entry in *reqArray* has a greater logical clock value, then the received request has already been satisfied and can be ignored. If the request is a new one, then node_i 's logical clock value is updated so that it is at least as great as the logical clock value contained in the REQUEST message - this is important for maintaining the total ordering of requests. The new request is then added to node_i 's *reqArray* and the message is broadcast to all of node_i 's neighbours not contained in the *alreadySeen* portion of the REQUEST message. Finally, if node_i owns the token but is not currently using the critical section, the token is given up - there is at least one node now requesting it.

Figure 6.6: HPRRouter/handle incoming traffic/incoming request (level 2)

Figure 6.7 shows the portion of the state diagram for HPRRouter which describes the handling of an incoming TOKEN message. The *elec* component of the TOKEN message is compared with Router_i's task identifier to determine whether or not node_i is the new owner

of the token. If node_i is the new owner of the token, then Invoker_i must be waiting to enter the critical section - the boolean flags are set and a GRANT message is sent to Invoker_i to inform it that the critical section is available. Let X be the newly elected node according to the token. If X is not node_i then node_i will forward the token to node_{via} , where node_{via} is the neighbour from which node_i initially received X 's request. The request being satisfied is then removed from *reqArray*.

Figure 6.7: HPRRouter/handle incoming traffic/incoming token (level 2)

Figure 6.8 shows the segment of the state diagram for HPRRouter which handles outgoing traffic. The outgoing message must be one of two types - a REQUEST message or a

TOKEN message. If the message is a REQUEST, then Router_i simply broadcasts the message to all of node_i's neighbours. If the message is the TOKEN message, then Router_i simply forwards the message to node_i's appropriate neighbour, as described in Figure 6.7. Node_i's logical clock is incremented to maintain the total ordering and the satisfied request is removed from *reqArray*.

Figure 6.8: HPRRouter/handle outgoing traffic (level 1)

6.3 A New Distributed Algorithm for Mutual Exclusion

The new algorithm - called HGB - uses messages similar to those defined in the RA algorithm ([18]) and local data structures similar to those employed in the HPR algorithm. Extra parameters are required in the messages to associate replies with requests in the general network (recall that the RA algorithm was designed for a complete network) and more data must be maintained locally than in the HPR algorithm because less control data is included in messages.

Although the new algorithm is similar in function to the RA algorithm, the HPR algorithm was a strong influence in the design of the algorithm. The new algorithm was actually written in an attempt to modify the HPR algorithm to avoid using messages which contain $O(n)$ information. Only after the algorithm was written were the similarities to the RA algorithm realized.

The HGB algorithm uses the same system of logical clocks and total ordering of critical section access requests as does the algorithm presented in [12]. The assumptions concerning communications between nodes in the network are the same as outlined for the HPR algorithm. There are two types of messages in the HGB algorithm: REQUEST messages and REPLY messages. A REQUEST message is of the form *REQUEST(requestor,time,sender)*, where *requestor* is the node identifier of the node requesting access to the critical section, *time* is the logical clock value associated with the request and *sender* is the node identifier of the last node to forward the message. A REPLY message is of the form *REPLY(requestor)* where *requestor* is the node identifier of the node requesting access to the critical section.

Each node wishing to enter the critical section sends a request message to each of its neighbours containing the node's identifier and current logical clock value. The node then waits until it receives a REPLY message containing its own identifier from each neighbour - it may then enter the critical section.

Node_i receiving a REQUEST(j,time_j,k) message must first determine whether or not it has already seen this request. If node_i has seen the request previously it replies immediately to node_k and ignores the request. If node_i has not seen the request before, the request is forwarded to node_i's neighbours excluding node_j and node_k (ie. the set neighbours_i-{node_j,node_k}) and sets the number of outstanding replies associated with the request to be the cardinality of the set to which the request was forwarded (ie. |neighbours_i-{node_j,node_k}|). Node_i then waits until it receives the required number of replies for node_j's request before sending a reply to node_k. When node_i has received the required number of requests it must determine whether or not the reply to node_k should be deferred. If node_i is currently in the critical section or has an outstanding request which precedes node_j's request in the total ordering, then node_i will defer the reply until it exits the critical section.

Node_i maintains the following local information:

- 1) *neighbours*. This is an array containing the set of nodes to which node_i is directly connected by communication links.
- 2) *clock*. This is node_i's logical clock which has an initial value of zero (0).
- 3) *reqTime*. This is the logical clock value used for Node_i's last critical section access request.

- 4) *osReplies*. This is the number of outstanding replies for Node_i's last critical section access request.
- 5) *requesting*. This is a boolean value indicating whether or not Node_i is currently executing or requesting access to the critical section.
- 6) *reqArray*. This is an array of request sets in which each set is associated with a specific neighbour of node_i. Each set contains the requests received by node_i which came directly from the associated neighbour. The elements of each set are records of type *reqInfo*. Figure 6.9 is a pascal-like description of the *reqInfo* record structure. Strictly speaking, the *reqSender* field in the *reqInfo* records is redundant since all elements in *reqArray[x]* came from *neighbour[x]*. However, the field is included to simplify the description of the algorithm.

```

reqInfo = record
    reqOrigin      : integer; /* Identifier of the request's originating node. */
    reqTime        : integer; /* Logical clock value for the request.          */
    reqSender       : integer; /* Identifier of the node which forwarded      */
                                /* this request to nodei.                      */
    reqOsReplies    : integer; /* Number of outstanding replies for request. */
    reqRepDeferred  : boolean; /* Indicates whether or not the reply for this */
                                /* request has been deferred by nodei or not. */
end

```

Figure 6.9: Structure of a *reqInfo* record.

The algorithm is presented below with each network node being described in terms of three processes sharing memory. The constant *me* contains the node's own identifier. The operations of set subtraction and addition are denoted by - and $\oplus$, respectively. Subfields of records are denoted using the notation *<record>.<field>*.

```

/* The shared data used by the three processes is listed below */
Integer  neighbours[1..m],          /* nodei can communicate directly with.... */
        clock initial(0),          /* logical clock */
        reqTime,                   /* logical clock value of my last request */
        osReplies;                 /* # of outstanding replies for last request */

Boolean requesting initial(false); /* is nodei currently requesting entry */

Semaphore sharedData;              /* for mutual exclusion on shared data */

reqInfoSet reqArray[1..m];        /* received request sets */

/* Invoke loops indefinitely executing the critical and noncritical section alternately. */
Process Invoke
  Repeat forever
    begin
      <noncritical section>

      P(sharedData);                /* beginning of entry protocol */
      requesting := true;
      clock := clock + 1;
      reqTime := clock;
      V(sharedData);

      osReplies := |neighbours|;
      send REQUEST(me, reqTime, me) to neighbours;
      waitfor(osReplies=0);

      <critical section>

      P(sharedData);                /* beginning of exit protocol */
      requesting := false;
      V(sharedData);
      ∀ r ∈ reqArray ∃ r.reqRepDeferred do
        begin
          send REPLY(r.reqOrigin) to r.reqSender;
          r.reqRepDeferred := false;
        end
      end
    end /* repeat */
  end /* Invoke */
end

```

[illegible]

6.3.1 Discussion of the New Algorithm

Lemma 1: All nodes receive and reply to all critical section access requests before the requestor enters the critical section.

Proof. A node requesting access to the critical section, sends REQUEST messages to all of its neighbours and awaits REPLYs from all of its neighbours. A node receiving a REQUEST message, forwards the message to any node it knows of which may not have seen the message - since it knows the requestor and the node from which it received the message, it forwards the REQUEST message to all other neighbours. Repeated requests are not forwarded but REPLYs are sent immediately. Ignoring the transmission links which carry REQUEST messages to any node, say node_j, such that the received REQUEST has been seen previously by node_j, the transmission links which the REQUEST travels over form a spanning tree for the network. The resulting message pattern can be thought of as a two-phase REQUEST/REPLY pattern. The copies of the REQUEST message travel outward from the requestor until the leaves of the spanning tree are reached and then the REPLY messages travel back from the leaves, following the spanning tree links (remembered using the *reqSender* field of the *reqInfo* data records). Therefore, all nodes must receive and reply to the REQUEST message before the requestor enters its critical section.

Lemma 2: The HGB Algorithm is safe.

Proof. Assume that two nodes, *i* and *j*, at some time are executing their critical sections concurrently. Without loss of generality, assume that $id_i < id_j$. By Lemma 1, we know that all nodes see all REQUEST messages and, eventually, send a REPLY message to all

REQUEST messages. Therefore, we know that node_i must have seen node_j's REQUEST message before node_j entered its critical section (and vice versa). There are two cases to consider: either node_i received the REQUEST message from node_j prior to requesting access to the critical section or node_i received the REQUEST message from node_j after requesting access to the critical section.

case I: Assume that node_i received node_j's REQUEST message before requesting access to the critical section. When node_i receives a REQUEST, node_i's logical clock is updated such that its value is at least $time + 1$, where $time$ is the logical clock value included in the REQUEST message. This implies that when node_i sends its REQUEST message, the logical clock value included in the message will be larger than $time$, causing node_j's request to precede node_i's request in the total order. When node_j receives node_i's REQUEST message, its local boolean *requesting* will be set to true since, by assumption, node_j must be either requesting access to or executing its critical section when node_i enters its critical section. Therefore, node_j will defer replying to node_i's REQUEST message when it is received. This means that node_i will not receive a REPLY message from node_j until after node_j exits its critical section. This means that node_i will not enter its critical section while node_j is in the critical section, contradicting the assumption.

case II: Assume that node_i received node_j's REQUEST message after requesting access to the critical section. If the logical clock value included in node_j's REQUEST message is less than node_i's *reqTime* value, then node_j will defer its REPLY message as described in case I. If the logical clock value in node_j's REQUEST message is less than or equal to

node_i's *reqTime* value, then node_i's request will precede node_j's in the total order since $id_i < id_j$. This means that node_i will now defer the reply to node_j since node_i's *requesting* value will be set to true. This is the same as case I due to symmetry. Again, the assumption that the two nodes execute the critical section concurrently is contradicted.

The algorithm can be shown to be free from deadlock and starvation. The proofs are as presented in [18] assuming that Lemma 1 holds. The algorithm gives slight priority to nodes with low identifiers because of the rule used in the total ordering but is basically fair. The algorithm makes no timing assumptions concerning events occurring at the participating nodes.

6.4 MTS Implementation of the New Algorithm

Let node_i be any node in the MTS model of a network executing the HGB algorithm. Figure 6.10 shows a tasks/alias diagram segment showing the components of node_i. The purpose of the *Invoker* and *Router* tasks is similar to that of the corresponding tasks in the implementation of the HPR algorithm. The *Invoker* alternates indefinitely between executing the critical and noncritical sections, using the entry and exit protocols each time it enters the critical section. The *Router* is node_i's interface to the rest of the network and processes incoming REQUEST messages. The *Reply* task is responsible for processing incoming REPLY messages. As in the implementation of the HPR algorithm, the task identifier of Router_i is used as the node identifier of node_i. Recall that the intention of this simulation was to study the messaging delays of the algorithm. Therefore, any overhead resulting from this multitasking implementation at each node is ignored. Also, communication between processes at node_i is assumed to be instantaneous.

In order to implement the HGB algorithm using this multitask structure, some new messages have been introduced. Reply_i will send a *GRANT* message to Invoker_i when the critical section becomes free after Invoker_i has issued a request. Invoker_i will send a *RELEASE* message to Router_i when exiting the critical section to inform Router_i that any deferred *REPLY* messages should now be transmitted.

Figure 6.10: Node Structure for the MTS Implementation of the HGB Algorithm.

Figure 6.11 shows the functional description of the Invoker task class, called *HGBInvoker*. Invoker_i simply executes the critical and noncritical sections indefinitely, using the protocols to enter and exit the critical section. Note that, because this algorithm does not utilize an explicit token, node_i must always distribute a request and wait for the replys before entering the critical section. Reply_i will inform Invoker_i when the critical section may be entered, based on receiving the appropriate number of *REPLY* messages, by sending a *GRANT* message.

Figure 6.11: HGBInvoker task class functional description.

Figure 6.12 shows the functional description of the Reply task class, called *HGBRepReceiver*. Whenever a REPLY message is received, Reply_i checks the parameter to determine whether or not the reply is for a request initiated by Invoker_i. If the reply is destined for node_i, then Reply_i will decrement node_i's *osReplies* counter, and determine whether or not any outstanding replies remain to be received. If there are no further replies to be received, then Reply_i will send a GRANT message to Invoker_i to inform it that the critical section is free. If the reply is not destined for node_i, then the appropriate entry in

node_i's *reqArray* is checked to determine whether or not node_i is still awaiting replies for the request which was previously forwarded by node_i. If no further replies are required, then node_i will either send a REPLY message to the node which originally sent the request to node_i or mark the reply deferred in *reqArray*.

Figure 6.12: HGBRepReceiver task class functional description.

Figures 6.13 to 6.18 show the hierarchically structured functional description of the Router task class, called *HGBRouter*. The figure captions show the MTS state machine editor label associated with each portion of the diagram to provide a frame of reference.

Figure 6.13 shows the top level of the state diagram for the Router task class. Router_i waits for either outgoing messages (the message sender was either Invoker_i or Reply_i, therefore the message is being sent out of the node) or incoming messages (the message

sender was neither $Invoker_i$ nor $Reply_i$). When it receives a message, it determines the direction (incoming or outgoing) and type of the message and handles it appropriately. The details associated with each type of message and each direction are shown in the following segments of the state diagram. At the top level, this diagram is exactly the same as the diagram for the Router task class used in the implementation of the HPR algorithm with the exception that, here, RELEASE messages are accepted at the receive state as well as REPLY and REQUEST messages.

Figure 6.13: HGBRouter (level 0)

Figure 6.14 shows the segment of the state diagram for the HGBRouter task class which deals with the handling of incoming messages. Incoming messages can be one of two types: either REQUEST messages or REPLY messages. The procedure for handling incoming requests is described later. If the message was an incoming reply, Router,

simply passes it on to Reply_i for processing.

Figure 6.14: HGBRouter/handle incoming traffic (level 1)

Figure 6.15 shows the segment of the state diagram for the HGBRouter task class which deals with the handling of incoming REQUEST messages. If the message is an incoming request, Router_i first removes any outdated requests made from the requestor. Since this is done every time a REQUEST message is received, there can never be more than one *reqInfo* record from a given requestor in *reqArray* at a given time. Next, Router_i must determine whether or not the request has been received previously from one of node_i's

other neighbours. If the request has been received previously, then node_i simply sends a REPLY message to the node that sent this copy of the request and ignores the request. If the request has not been seen by node_i , then Router_i will update *clock*, ensuring that its value is larger than the timestamp^L included in the REQUEST message - this is important for maintaining the system of logical clocks. Router_i will then broadcast the request to the set of node_i 's neighbours, ignoring the request originator and the node from which node_i received the message (ie. $\text{neighbours}_i - \{\text{requestor}, \text{sender}\}$). The number of replies required for this forwarded request is computed and the new request is stored in *reqArray*. If $|\text{neighbours}_i - \{\text{requestor}, \text{sender}\}| = 0$, then node_i must either send a reply to the node from which the request came or mark the reply deferred in *reqArray*.

Figure 6.15: HGBRouter/handle incoming traffic/handle REQUEST (level 2)

Figure 6.16 shows the portion of the state diagram for the HGBRouter task class which describes the handling of outgoing messages. The message received must be one of three types: REQUEST, REPLY or RELEASE. The procedures for handling REQUEST and RELEASE messages are presented later. If the message is a reply, then Router_i simply sends the reply to the node from which the associated request was originally received.

Figure 6.16: HGBRouter/handle outgoing traffic (level 1)

Figure 6.17 shows the segment of the state diagram for HGBRouter which describes the handling of outgoing REQUEST messages. Since Router_i directly processes incoming

REQUEST messages, an outgoing request is one that is being initiated by Invoker_i . Therefore, Invoker_i has already set all of the local shared variables related to the request and all that remains is for Router_i to forward the request to all of node_i 's neighbours.

Figure 6.17: HGBRouter/handle outgoing traffic/outgoing REQUEST (level 2)

Figure 6.18: HGBRouter/handle outgoing traffic/send deferred REPLYs (level 2)

Figure 6.18 shows the segment of the state diagram for the HGBRouter task class which describes the procedure for handling RELEASE messages. A RELEASE message is sent to Router_i by Invoker_i when the Invoker exits its critical section. The message informs Router_i that Invoker_i is out of the critical section and all deferred replies should now be sent. The Router simply scans *reqArray* and sends replies for all requests marked deferred.

6.5 Simulation Parameters and Results

Simulation studies were performed for the two algorithms using the assumptions that: 1) the transmission time for a given message is proportional to the amount of information contained in the message and 2) the average time between critical section access requests is the same for all nodes in the network. Specifically, the studies were carried out using a deterministic communication delay function which set the transmission time for a message to be $n+1$ time units, where n is the number of parameters contained in the message. If it is assumed that the message type information, the logical clock values and the network node identifiers all require approximately the same number of bits then this assumption is reasonable. Table 6.1 shows the transmission times, calculated using this assumption, for each type of message used in the algorithms.

<u>algorithm</u>	<u>message</u>	<u>parameters</u>	<u>transmission time</u>
HPR	REQUEST	$3+k, 2 \leq k \leq n$	$4+k, 2 \leq k \leq n$
	TOKEN	$1+n$	$2+n$
HGB	REQUEST	3	4
	REPLY	1	2

Table 6.1: Table of message transmission times.

Simulation runs were done to collect data on the time delay between requesting access to the critical section and entering the critical section. The average rate of critical section

access requests was varied by increasing or decreasing the average delay in the noncritical section. By increasing the average time spent in the noncritical section, the average rate of requests in the network is lowered. Each algorithm was simulated on three different network models: a tree network composed of five nodes, a complete network composed of five nodes and a general network composed of eight nodes. The tree network and general network are shown in Figure 6.19.

Figure 6.19: The tree and general network maps.

The noncritical section delays are generated using uniformly distributed pseudorandom variables on the range, 0 to 2μ time units, where μ is the distribution's mean. All simulations were performed using uniformly distributed critical section delays, at all nodes, in the range, 0 to 10 time units. Figures 6.20, 6.21 and 6.22 show the average delay to enter the critical section and percentage of time that some node is using the critical section plotted against the average noncritical section delay, for the tree, complete and general networks, respectively.

In all cases, the simulations of the HGB algorithm resulted in lower entry protocol delays than did the simulations of the HPR algorithm, although the results are very similar for the tests on the complete network. The complete network and tree network studies provide an interesting insight into the functioning of the two algorithms as the density of communication links increases. The HPR algorithm, through the use of control

information in the REQUEST messages, restricts the number of messages sent through the network. Recall that a REQUEST message in the HPR algorithm contains the identifiers of all nodes that have seen or are about to see the REQUEST. Thus, in a complete network, a node requesting access to the critical section sends a message to every other node in the network directly and one of those nodes will (eventually) send the token back. The HGB algorithm, on the other hand, takes a more straightforward approach to routing requests through the network - every node sends the request to all neighbours except the node that it came from and the original requestor. Thus, with the HGB algorithm, the number of messages being sent through the network increases proportionally with the number of communication links in the network. Although the complete network causes the HGB algorithm to send more messages than the tree network, the increased communication capacity of the complete network does result in lower average delays for the algorithm. As will be shown in more detail when the asymptotic bit and message complexities are presented, the complete network is the worst case for the HGB algorithm and the best case for the HPR algorithm.

The critical section utilization graphs are interesting because they seem to show, for all three networks, the network capacity for critical section access requests being reached when the HPR algorithm is used. This is shown as a "plateau" on the critical section utilization graphs at the higher request rates. This phenomenon is seen using the HGB algorithm only in the tree network, although this is the only case in the studies where utilization actually drops as the request rate increases. The simulation results show that node 3 (the centre of the tree network) actually achieved a higher average critical section access rate when the average delay in the noncritical section was 15 time units than when this delay was 5 time units. This indicates that at the highest request rate the communication links, especially near the centre of the network, were saturating to the point of decreasing critical section usage.

Figure 6.20: The tree network graphs.

Figure 6.21: The complete network graphs.

Figure 6.22: The general network graphs.

It is possible to calculate estimates for the average entry protocol delay when the traffic rate is very low. By very low, we mean that the number of requests in the network is so low that when a request is issued, we can expect that there will be no delay introduced by message queues or by other nodes being in the critical section. This can be used to estimate a lower limit on the expected delays for each algorithm in light traffic. Table 6.2 shows these estimates for each of the networks using the two algorithms. These limits indicate that for each network, the nodes will continue to achieve lower average entry protocol delays using the HGB algorithm as the request traffic rate decreases. The method used to derive these estimates is described below.

<u>network</u>	<u>HGB avg</u>	<u>HPR avg</u>
tree	15.6	20.64
complete	12	12.8
general	21	26.13

Table 6.2: Minimum average delay estimates for the test networks with each algorithm.

When a node requests access to the critical section using the HPR algorithm, the minimum delay possible for that given request depends on the distance the requestor is from the node which holds the token. Assuming that there is no other request in the network, the delay will be the amount of time it takes to transmit the REQUEST message to the token owner and then transmit the token to the requestor. With transmission delays computed according to the delay function used in the simulations, we can compute a matrix showing the minimum transmission time for a REQUEST message to travel between every pair of nodes in the network. Figure 6.23(a) shows the matrix computed for the tree network. The (i,j) element of the matrix gives the minimum transmission time for a REQUEST which originates at node _{i} to travel to node _{j} . Note that due to the augmentation of REQUEST messages' control information, as the request propagates through the network, the matrix will not, in general, be symmetric. Figure 6.23(b) is a matrix in which the (i,j) element

represents the minimum transmission time taken to send the token to node_i from node_j in the tree network. By adding these two matrices and taking the mean of the resulting matrix elements, we can compute an estimate of the minimum average delay for requests in the network using the HPR algorithm.

0	14	6	14	23	0	14	7	14	21
14	0	6	14	23	14	0	7	14	21
8	8	0	8	17	7	7	0	7	14
16	16	7	0	7	14	14	7	0	7
22	22	13	6	0	21	21	14	7	0

(a)
(b)

Figure 6.23: Low traffic estimate matrices for tree network using the HPR algorithm.

Using the HGB algorithm, a low traffic minimum average delay estimate can be computed by calculating, for each node, the minimum transmission times for the longest path on the spanning tree followed by the REQUEST messages. Note that if the leaf of the spanning tree actually has neighbours in the network, the path travelled by REQUEST messages will be one link longer than the actual path in the tree. This is due to the fact that the leaf will still forward the REQUEST but all of its neighbours will have seen it before and simply reply to it. Table 6.3 shows the longest path minimum transmission time associated with each node in the tree network, for both REQUESTs and REPLYs. The mean of the sum of these columns gives the low traffic average delay estimate for the HGB algorithm in the tree network.

<u>node</u>	<u>REQUEST delays</u>	<u>REPLY delays</u>
1	12	6
2	12	6
3	8	4
4	8	4
5	12	6

Table 6.3: Longest path min. transmission time for HGB algorithm in the tree network.

6.6 Comparison of Asymptotic Message and Bit Complexities

In a general network, the HPR algorithm requires at most $2e$ REQUEST messages to transmit a request through the network, where e is the number of communication links in the network ($n-1 \leq e \leq n(n-1)/2$). A request will traverse a communication link at most twice but this is a pessimistic bound due to the control information included in the REQUEST messages by this algorithm. Transmitting the token requires at most $n-1$ messages. It is easy to see the message complexity of this algorithm for some special cases: complete network, tree network and ring network. In a complete network, exactly $n-1$ REQUEST messages and 1 TOKEN message are required in order for a node to enter the critical section. In a ring network, at most $n+1$ REQUEST messages and $n-1$ TOKEN messages are required to request access to the critical section. In a tree network, exactly $n-1$ REQUEST messages and at most $n-1$ TOKEN messages are required to request access to the critical section.

The HGB algorithm uses at most $2e$ REQUEST and $2e$ REPLY messages, where e is the number of communication links in the network ($n-1 \leq e \leq n(n-1)/2$).

The HPR algorithm uses two types of messages: REQUESTs and TOKENs. A request has the form, $\text{REQUEST}((\text{reqId}, \text{reqTime}), (\text{sender}, \text{alreadySeen}))$, and requires at most $1+b+(n+2)\log n$ bits, assuming the message type requires 1 bit, b is the number of bits required to store a logical clock value and the node identifiers are stored using $\log n$ bits. The token is sent in a message of the form, $\text{TOKEN}(\text{elec}, \text{lud})$, and requires $1+\log n+nb$ bits, using the same assumptions.

The HGB algorithm uses two types of messages: REQUESTs and REPLYs. A request has

the form, REQUEST(reqId, reqTime, sender) and requires $1+2\log n+b$ bits, assuming the message type requires 1 bit, b is the number of bits required to store a logical clock value and the node identifiers are stored using $\log n$ bits. The replies are sent using messages of the form, REPLY(reqId) and require $1+\log n$ bits using the same assumptions.

Table 6.4 shows the asymptotic message and bit complexities for the general case and special cases of tree, complete and ring networks using the two algorithms. Notice that the HGB algorithm achieves message complexities as good as the HPR algorithm in all cases except the complete network. Whereas the message and bit complexities of the HPR algorithm may be unduly pessimistic in the general case, the bound given for the HGB algorithm is probably realistic. However, looking at the bit complexity of the algorithms, the HGB algorithm achieves an $O(n)$ improvement in the the general case and in the special cases of the ring and tree networks. The bit complexity for the complete network shows that the HPR algorithm may be slightly better than the HGB algorithm.

<u>Algorithm</u>	<u>Topology</u>	<u>Message Complexity</u>	<u>Bit Complexity</u>
HGB	tree	$O(n)$	$O(n(\log n+b))$
	ring	$O(n)$	$O(n(\log n+b))$ *
	complete	$O(n^2)$	$O(n^2(\log n+b))$
	general	$O(e), (n-1) \leq e \leq n(n-1)/2$	$O(e(\log n+b))$
HPR	tree	$O(n)$	$O(n^2(\log n+b))$
	ring	$O(n)$	$O(n^2(\log n+b))$
	complete	$O(n)$	$O(n^2 \log n + nb)$ *
	general	$O(e), (n-1) \leq e \leq n(n-1)/2$	$O(en(\log n+b))$

* only one token message sent for each critical section access request

Table 6.4: Asymptotic bit and message complexities.

6.7 Summary

This chapter has presented two algorithms which achieve distributed mutual exclusion in arbitrary, connected networks. The results of simulation studies done using MTS and the

asymptotic complexities for the two algorithms were also presented.

According to the simulation studies done, the new algorithm presented in this chapter achieves lower average entry protocol delays than does the HPR algorithm. The exception to this is the complete network where, below a certain request rate, the average delays are identical.

The bit complexities of the two algorithms also indicate that the new algorithm achieves better results than does the HPR algorithm. In the general case, as well as in the special case of tree and ring networks, the new algorithm improves on the bit complexity of the HPR algorithm by $O(n)$. In the complete network, the HPR algorithm may have slightly better bit complexity, depending on the relative size of the logical clock values and node identifiers used.

Both the simulation study and the asymptotic complexities of these algorithms indicate clearly that the relative performance of the new algorithm is best in networks with sparse communication links while the relative performance of the HPR algorithm is best in dense networks. Considering the difficulty and expense involved in building networks with dense interconnections, the relative performance of the new algorithm in sparse networks is interesting.

Recall that these simulation results only take communication delays into account. Both algorithms require each node to perform $O(n)$ processing whenever a message is received. Therefore, it would be interesting to add some processing delays into the simulation to study the effects. The message transfer control information used by the HPR algorithm may, in this case, cause it to outperform the HGB algorithm in dense networks.

Chapter 7

Conclusions and Suggestions for Further Research

This thesis has presented MTS, a discrete-event simulator for studying multitasking and distributed systems, and its graphical model development environment. MTS uses a model of Communicating State Machines (CSMs) combined with the notion of task classes to provide a system which supports modularity and encourages the development of model and task class libraries. The resulting system has been found capable of modeling complex systems in detail.

To date, MTS has been in development since April 1987. The simulation engine, in an earlier version, was originally created by Professor John Neilson and Dorina Petriu. Starting with that version of MTS in September 1987, the event handling was redesigned using typed states such that the passage of time would be accounted for solely in states. By October 1987, the event handling had been rewritten and the processor, processor pool and communication network models, as described in Chapter 3, had been incorporated into MTS. The implementation of the graphical user interface for MTS began in October 1987 and was completed by February 1988. Thus, MTS represents somewhat less than a man-year of work. It is believed that the rapid development of MTS was possible largely because of the choices made concerning the programming languages used to implement the system. The simulation engine was written in C which provides the performance needed to carry out the computations required by simulation. Smalltalk-80 was chosen for the implementation of the model development environment because of the excellent facilities provided for both the development and use of graphical interfaces.

The study of distributed mutual exclusion, which originally focused on a modified version of the RA algorithm and the HPR algorithm in a tree network, led to the development of a new algorithm for arbitrary, connected networks which has been shown to outperform the HPR algorithm, especially in sparsely connected networks. The models developed for the study of distributed mutual exclusion presented in Chapter 6 of this thesis were created and debugged using approximately one man-week of work each.

The following is a brief list of topics, related to this thesis, which would be of interest for further study:

- 1) Enhancement of the debugging facilities provided with MTS including the development of an animator. Currently, the task of debugging an MTS model involves sifting manually through lengthy listings of output from the simulation engine. Probably the easiest way to animate an MTS model would be to simulate the model directly in the Smalltalk-80 environment. The cost of this would be the loss of performance involved in moving the execution of simulations from compiled C to interpreted Smalltalk-80. Another alternative would be to duplicate the simulation engine - use Smalltalk-80 for debugging and C once the model is ready for intensive simulation. Recalling that MTS assumes the presence of user-defined functions to implement a model, the problem is then to choose a suitable language for the specification of the user-defined functions required by the models.
- 2) Development of an abstraction mechanism which could be used, in the context of state diagrams and task/alias diagrams, to support top-down refinement along with the current bottom-up clustering facility.

- 3) The use of application specific model development interfaces could be explored. The current model definition interface for MTS was designed to be useable in a wide variety of applications. For that reason, modeling flexibility has been stressed throughout. At the same time, however, the model definition interface and the task class definition interface has been kept separate so that other model development systems could be contemplated at a later date. For instance, a model definition interface designed specifically for systolic architectures would probably be quite easy to develop. One task class could represent the activities performed by each systolic processor while another task class could encode the algorithm to be executed on the array. The function of the model configuration interface would then be to describe the structure of the array and initialize the algorithm.

References

- [1] Borning, A., "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory", *ACM Transactions on Programming Languages and Systems*, Vol. 3, No. 4, October 1981, pp. 353-387.
- [2] Borning, A., "Defining Constraints Graphically", *ACM Computer Human Interaction, CHI'86 Conference Proceedings*, April 1986, pp. 137-143.
- [3] Buhr, R. J. A., *System Design with Ada*, Prentice-Hall, Englewood Cliffs, New Jersey, 1984.
- [4] Carvalho, O. S. F. and Roucairol, G., "On Mutual Exclusion in Computer Networks", *Communications of the ACM*, Vol. 26, No. 2, February 1983, pp. 146-147.
- [5] Dähler, J., Gerber, P., Gisiger, H.-P. and Kündig, A., "A Graphical Tool for the Design and Prototyping of Distributed Systems", *ACM SIGSOFT Software Engineering Notes*, Vol. 12, No. 3, July 1987, pp. 25-34.
- [6] Delisle, N. and Schwartz, M., "A Programming Environment for CSP", *ACM SIGSOFT/SIGPLAN 2nd Software Engineering Symposium on Practical Software Development Environments*, Palo Alto, California, 1986, pp. 34-41.

-
- [7] Genrich, H. J., "Predicate/Transition Nets", *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Vol. I*, Springer-Verlag, Berlin, 1986, pp. 207-247.
 - [8] Harel, D., "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, Vol. 8, 1987, pp. 231-274.
 - [9] Helary, J. M., Plouzeau, N. and Raynal, M., "A Distributed Algorithm for Mutual Exclusion in an Arbitrary Network", Technical Report No. 281, IRISA, Université de Rennes, France, January 1986.
 - [10] Hoare, C. A. R., "Communicating Sequential Processes", *Communications of the ACM*, Vol. 21, No. 8, August 1978, pp. 666-677.
 - [11] Kramlich, D., Brown, G. P., Carling, R. T. and Herot, C. F., "Program Visualization: Graphics Support for Software Development", *Proceedings of the ACM/IEEE 20th Design Automation Conference*, June 1983, IEEE-CS Press, Los Alamitos, California, pp. 143-149.
 - [12] Lamport, L., "Time, Clocks and the Ordering of Events in a Distributed System", *Communications of the ACM*, Vol. 21, No. 7, July 1978, pp. 558-565.
 - [13] London, R. L. and Duisberg, R. A., "Animating Programs Using Smalltalk", *Computer*, August 1985, pp. 61-71.
 - [14] Martin, A. J., "Distributed Mutual Exclusion on a Ring of Processes", *Science of Computer Programming*, Vol. 5, 1985, pp. 265-276.

-
- [15] Peterson, J. L., "Petri Nets", *Computing Surveys*, Vol. 9, No. 3, September 1977, pp. 223-252.
- [16] Powell, M. L. and Linton, M. A., "Visual Abstraction in an Interactive Programming Environment", *SIGPLAN Symposium on Programming Language Issues in Software Systems, SIGPLAN Notices*, Vol. 18, No. 6, ACM, June 1983, pp. 14-21.
- [17] Reiss, S. P., "GARDEN Tools: Support for Graphical Programming", *Advanced Programming Environments: Proceedings of an International Workshop*, Trondheim, Norway, June 16-18, 1986, Springer-Verlag, pp. 59-72.
- [18] Ricart, G. and Agrawala, A. K., "An Optimal Algorithm for Mutual Exclusion in Computer Networks", *Communications of the ACM*, Vol. ~~24~~ 24, No. 1, January 1981, pp. 9-17.
- [19] Rockström, A. and Saracco, R., "SDL - CCITT Specification and Description Language", *IEEE Transactions on Communications*, Vol. COMM-30, No. 6, June 1982, pp. 1310-1318.
- [20] Sutherland, I., "Sketchpad: A Man-Machine Graphical Communication System", *AFIPS Proceedings - Spring Joint Computer Conference*, 1963, pp. 329-346.
- [21] Suzuki, I. and Kasami, T., "A Distributed Mutual Exclusion Algorithm", *ACM Transactions on Computer Systems*, Vol. 3, No. 4, November 1985, pp. 344-349.

-
- [22] Weide, B. W., Brown, M. E., Alegria, J. A. S. and Meyer, G., "A Graphical Interconnection Language and its Application to Concurrent and Real-Time Programming", *Proceedings of the 20th Annual Allerton Conference on Communication, Control and Computing*, University of Illinois, IEEE, October 1982, pp. 567-576.

Appendix A

Format of Generated Source Files

Figure A.1 shows the general format used for the source code files generated by MTS. Figures A.2 to A.4 describe a simple model in which a sending task determines probabilistically whether or not to rendezvous again with a receiver each time it receives an acknowledgement from the receiver. The sender has a parameter called STOPPROB which gives the probability of stopping the process at each attempt. The model contains one task of each type and the sender uses the alias "RECEIVER" for the receiving task. The generated source code file for this model is then shown.

Standard "#include" references required to access simulator data structures. Required for access to task state tables, processor pool and processor tables, collection support data structures, etc.
User's comment and any additional "#include" or "#define" statements required for the model. This text is replicated directly from the top, right-hand view of the main model editor.
Initialization of the functional definition of the elements of the class set. This is contained in a procedure called "init_sim()". Each class definition is encoded as a set of table entries which describe the structure of the class' state machine.
User-defined functions which are required to complete the definition of the classes. <ul style="list-style-type: none">• arc selection functions• assignment functions• delay functions• process delay functions• dynamic initialization functions
Communication network delay and message delivery success functions (if other than the default is to be used).
main procedure: <ul style="list-style-type: none">• set the communication network parameters (if not default)• initialize the simulation (including class definitions)• seed the random number streams• create processor pools and initial tasks• initialize task instance variables (including aliases)• call the main simulation routine

Figure A.1: Generated source code file layout.

Figure A.2: Sender class functional definition.

Figure A.3: Receiver class functional definition.

Figure A.4: Static task/alias diagram for the model.

```

/* standard includes for mts simulation model. */
#include "mts.h"
#include "class.h"
#include "mess.h"
#include "task.h"
#include "stats.h"
#include "processor.h"
#include "channel.h"
#include "collection.h"

/* useful external procedures */
extern int true(), timeout(), notimeout(); /* arc selection procedures */
extern int alwaysDeliver(); /* communication channel transmission success */
extern double zero(); /* communication channel delay function */

/* simulation initialization routines */
extern int task_create();
extern int create_pool();
extern void set_channel_parms();

/*
This is a model of a system in which a ReqSender task sends REQ messages to
a ReqReceiver task stochastically. The ReqSender has a parameter called STOPPROB
which gives the probability of terminating the transmission of messages on a given
trial. At least one REQ message is sent and acknowledged by the ReqReceiver.
*/

init_classes()
{
    extern int senderArc1Assign(), receiverArc1Assign(), senderArc3Assign();
    extern int senderArc2Select();
    extern int senderArc2Assign();

    /* class definition for: receiver, Smalltalk class: ReqReceiver */
#define FROM -1
    class_tab[0].name = "receiver";
    class_tab[0].statep = &state_tab[0];
    class_tab[0].arcp = &arc_tab[0];

    entry_tab[0].entryId = 1000;
    entry_tab[0].name = "REQ";

    entry_tab[1].entryId = 1001;
    entry_tab[1].name = "ACK";

    state_tab[0].classid = 0;
    state_tab[0].name = "?REQ";
    state_tab[0].arcp = &arc_tab[0];
    state_tab[0].state_type = RECEIVE;
    state_tab[0].entryp = &entry_list_tab[0];

    entry_list_tab[0].entryPtr = &entry_tab[0];
    entry_list_tab[0].next = NULL_POINTER;

```

```

arc_tab[0].name = "FROM?REQ/ FROM!ACK";
arc_tab[0].tstate = 0;
arc_tab[0].aflag = TRUE;
arc_tab[0].afun = receiverArc1Assign;
arc_tab[0].sflag = TRUE;
arc_tab[0].rtask = FROM;
arc_tab[0].rentry = &entry_tab[1];
arc_tab[0].sfun = true;

#undef FROM
/* class definition for: sender, Smalltalk class: ReqSender */
#define COUNT -1
#define RECEIVER -2
#define STOPPROB -3
#define STOPSTREAM -4
class_tab[1].name = "sender";
class_tab[1].statep = &state_tab[1];
class_tab[1].arcp = &arc_tab[1];

entry_tab[2].entryId = 1000;
entry_tab[2].name = "REQ";

entry_tab[3].entryId = 1001;
entry_tab[3].name = "ACK";

state_tab[1].classid = 1;
state_tab[1].name = "";
state_tab[1].arcp = &arc_tab[3];
state_tab[1].state_type = CONNECT;

state_tab[2].classid = 1;
state_tab[2].name = "RECEIVER?ACK";
state_tab[2].arcp = &arc_tab[2];
state_tab[2].state_type = RECEIVE;
state_tab[2].stask = RECEIVER;
state_tab[2].entryp = &entry_list_tab[1];

entry_list_tab[1].entryPtr = &entry_tab[3];
entry_list_tab[1].next = NULL_POINTER;

state_tab[3].classid = 1;
state_tab[3].name = "";
state_tab[3].state_type = CONNECT;

arc_tab[1].name = "RECEIVER?ACK &test to stop/print COUNT; stop simulation";
arc_tab[1].tstate = 2;
arc_tab[1].aflag = TRUE;
arc_tab[1].afun = senderArc1Assign;
arc_tab[1].sfun = true;

arc_tab[2].name =
    "RECEIVER?ACK&test not to stop/COUNT++; RECEIVER!REQ";
arc_tab[2].tstate = 1;
arc_tab[2].aflag = TRUE;

```

```

arc_tab[2].afun = senderArc2Assign;
arc_tab[2].sflag = TRUE;
arc_tab[2].rtask = RECEIVER;
arc_tab[2].rentry = &entry_tab[2];
arc_tab[2].sfun = senderArc2Select;
arc_tab[2].narc = &arc_tab[1];

arc_tab[3].name = "/COUNT=1; RECEIVER!REQ";
arc_tab[3].tstate = 1;
arc_tab[3].aflag = TRUE;
arc_tab[3].afun = senderArc3Assign;
arc_tab[3].sflag = TRUE;
arc_tab[3].rtask = RECEIVER;
arc_tab[3].rentry = &entry_tab[2];
arc_tab[3].sfun = true;

#undef COUNT
#undef RECEIVER
#undef STOPPROB
#undef STOPSTREAM

}
#define REQ 1000
#define ACK 1001

/* functions for class: receiver */
#define FROM tp->var[1]
int receiverArc1Assign(tp)
struct tcb *tp; /* task pointer */
{
    FROM = (double) tp->mfrom;
}

#undef FROM

/* functions for class: sender */
#define COUNT tp->var[1]
#define RECEIVER tp->var[2]
#define STOPPROB tp->var[3]
#define STOPSTREAM tp->var[4]
int senderArc1Assign(tp)
struct tcb *tp; /* task pointer */
{
    printf("number of rendezvous: %g\n", COUNT);
    quitf = TRUE; /* global simulator flag */
}

int senderArc2Assign(tp)
struct tcb *tp; /* task pointer */
{
    COUNT = COUNT + 1.0;
}

```



```

int senderArc2Select(tp)
struct tcb *tp;                                /* task pointer */
{
    extern double uniform();

    double value;

    value = uniform(0.0,1.0,STOPSTREAM);
    return(value > STOPPROB);
}

int senderArc3Assign(tp)
struct tcb *tp;                                /* task pointer */
{
    COUNT = 1.0;
}

#undef COUNT
#undef RECEIVER
#undef STOPPROB
#undef STOPSTREAM
#undef REQ
#undef ACK
main()
{
    extern void init_sim(), run_sim();
    extern int seed();
    extern int index_of_named_class(), index_of_named_task();

    int pid, tid;                               /* processor and task ids */

    init_sim();

    seed(12345, 0);
    tid = task_create(index_of_named_class("sender"), "send", NULL_ID, NULL_POOL, 0);
    tid = task_create(index_of_named_class("receiver"), "rcv", NULL_ID, NULL_POOL, 0);

    tid = index_of_named_task("send");
    task_tab[tid].var[4] = 0.0;
    task_tab[tid].var[2] = (double) index_of_named_task("rcv");
    task_tab[tid].var[3] = 0.1;

    run_sim(100.0);
}

```

In the initialization of the task instance variables, notice that the variable names used in the graphical interface are mapped into an array in the task's control block. Examining the "#define" statements used in the definition of the sender class, we see that var[2] is RECEIVER, var[3] is STOPPROB and var[4] is STOPSTREAM.

Appendix B

Graphical Editing Modes and Operations

This appendix will present the editing modes and the editing operations associated with each for the main graphical editors used in MTS. The first section will present the modes and operations used in the editing of state diagrams. The second section will present the modes and operations used in the editing of static task/alias diagrams.

B.1 State Machine Editing Modes in MTS

Some of the editing modes described in this section are applicable to either state operations or arc operations. For these modes, the setting of the "state(s)" and "arc(s)" soft buttons are used to determine which type of operation is to be performed. The terms "state mode" and "arc mode" will be used to indicate the settings of these soft buttons throughout this section.

B.1.1 "add arc/state" mode

In state mode, the user can add states to the state machine by clicking the red button on the graphic view of the state machine. The point indicated by the red button click will be the state's centre point. A menu will be presented asking the user to select a state type (one of connect, delay, process or receive). When the user has made a selection from the menu, a new state of the indicated type will be created and displayed.

In arc mode, the user can add arcs to the state machine by using the red button to indicate the following information (in the order given):

- 1) The arc's originating state: click the red button when the cursor is positioned inside the desired state.
- 2) The arc's terminating state: click the red button when the cursor is positioned inside the desired state.
- 3) An optional path for the directed edge from the originating state to the terminating state can be entered using the red button while the cross hair cursor is displayed. Entry of optional path points may be terminated by clicking either the yellow or blue button.

The newly created arc will be displayed in the graphic view of the state machine using either a rubber-banded edge, a spline or a straight line, as appropriate.

B.1.2 "add arc using labels" mode

This mode is only applicable to arcs. This mode is provided to allow arcs to be added to the state machine when the originating and terminating states do not fit on the screen at the same time. New arcs may be added to the state machine by clicking the red button on the desired originating state. A menu showing the labels of possible terminating states will then be displayed from which the user may select the desired terminating state of the arc. No option for specifying an alternate path for the arc is provided in this mode.

B.1.3 "delete arc/state" mode

In state mode, states may be deleted from the state machine by clicking the red button while the cursor is positioned inside the state to be deleted. All arcs incident on the specified state

will also be removed.

In arc mode, arcs may be deleted by clicking the red button while the cursor is located on the arc in the graphic view of the state machine.

B.1.4 "move arc/state" mode

In state mode, states may be moved from one location in the graphic view of the state machine to another. The state to be moved is first selected by clicking the red button while the cursor is positioned inside the state. A new centre point for the state is then specified by clicking the red button again.

In arc mode, arcs may have new paths specified from the arc's originating state to the arc's terminating state. The arc is selected by clicking the red button while the cursor is positioned on the arc. Once the arc has been selected, then any old optional points for the arc are discarded and the system waits for the user to enter a new set of points to specify the path. Not specifying any optional points once the arc is selected will cause the arc to be represented by a straight line.

B.1.5 "cluster states" mode

This mode is used to create a new level in the hierarchical structure of the state machine. The user specifies a rectangle on the screen, using the red button to specify the upper left corner and the lower right corner, which contains the states to be moved down a level in the hierarchy. These states will be replaced in the current level of the state machine by a single cluster state.

B.1.6 "expand a cluster" mode

This mode is used to reverse a cluster operation. The user specifies a cluster state to be expanded by clicking the red button while the cursor is positioned inside the desired state. The state will be replaced by the set of states it represents in the next lower level of the state machine hierarchy.

B.1.7 "move arc point" mode

This mode provides an easy method for neatening the graphical view of a state machine. An arc point is selected by positioning the cursor close to the point while the red button is pressed. Once a point is selected, the arc segments for which the selected point is an end point will be erased and redrawn using grey lines. While the red button is pressed the point can be moved and the graphic view of the state machine will be updated until the red button is released. The last position of the cursor before the red button is released is used as the new endpoint for the two arc segments.

B.1.8 "set arc/state parms" mode

In either arc or state mode, an object may be specified using the red button and a new editor will be created which will enable the user to set parameters which are relevant to the specified object.

B.2 Static Task/Alias Diagram Editing Modes in MTS

Some of the editing modes described in this section are applicable to either task operations or alias operations. For these modes, the setting of the "task(s)" and "arc(s)" buttons are used to determine which type of operation is to be performed. The terms "task mode" and "alias mode" will be used to indicate the setting of these soft buttons throughout this section.

B.2.1 "add task/alias" mode

In task mode, the user can add tasks to the model by clicking the red button on the graphic view of the model. The point indicated by the position of the cursor will be the centre point for the rectangle representing the task on the display. A menu will be presented listing the class names in use in the model. Once the user has selected a class, a new instance of the specified class will be created and displayed.

In alias mode, the user can add aliases to the model by using the red button to indicate the following information (in the order given):

- 1) The originating task of the alias: click the red button when the cursor is positioned on the desired task.
- 2) The terminating task of the alias: click the red button when the cursor is positioned on the desired task.
- 3) An optional path for the directed edge from the originating task to the terminating task may be entered using the red button if there are not already aliases defined from the originating task to the terminating task. If there are already aliases defined, the path used for those will be used for the new one as well. If no previous alias

exists, entry of optional points may be terminated by clicking either the yellow or blue button.

- 4) The alias label must be specified. The system will present a menu with all the variable names from the class definition of the originating task. The user will select a name from the menu.

The newly created alias will be displayed in the task/alias diagram of the model using either a straight line or a rubber-banded edge, as appropriate.

B.2.2 "add alias using labels" mode

This mode is only applicable to aliases. The mode is provided to allow aliases to be added to the task/alias diagram when the originating and terminating tasks do not fit on the screen at the same time. New aliases may be added to the model by clicking the red button on the desired originating task. A menu showing the names of possible terminating tasks will then be displayed from which the user may select the desired terminating task. Once a terminating task has been selected, a menu will be presented from which the user may select the label for the alias. No option for specifying an alternate path for the alias is provided in this mode.

B.2.3 "delete task/alias" mode

In task mode, tasks may be deleted by clicking the red button while the cursor is positioned on the task to be deleted. All incident aliases will be deleted along with the task.

In alias mode, aliases may be deleted by clicking the red button while the cursor is positioned on the alias to be deleted. If the directed edge selected by the user represents

more than one alias, the user will be asked whether or not all of the aliases should be deleted. If the user answers "no", then the user will be asked to select a specific alias from a menu. The selected alias(es) will be deleted.

B.2.4 "move task/alias" mode

In task mode, tasks may be moved from one location to another in the task/alias diagram. A task is selected by clicking the red button while the cursor is positioned inside the task. A new centre for the selected task may then be specified by clicking the red button again.

In alias mode, aliases may have new paths specified from the originating task to the terminating task. The alias may be selected by clicking the red button while the cursor is positioned on the arc. Then an optional path may be specified for the arc.

B.2.5 "cluster tasks" mode

This mode is used to create a new level in the hierarchical structure of the task/alias diagram. The user specifies a rectangle on the screen, using the red button to specify the upper left corner and then the lower right corner, which contains the tasks to be moved down a level in the hierarchy. These tasks will be replaced in the current level of the model by a single cluster task.

B.2.6 "expand a cluster" mode

This mode is used to reverse a cluster operation. The user specifies a cluster task to be refined by clicking the red button while the cursor is positioned on the task. The task will

be replaced by the set of tasks it represents in the next lower level of the task/alias diagram hierarchy.

B.2.7 "move alias point" mode

This mode provides a easy method for neatening the task/alias diagram. An alias point is selected by positioning the cursor close to the point while the red button is pressed. Once a point is selected, the alias segments for which the selected point is an end point will be erased and redrawn using grey. While the red button is pressed the point can be moved and the task/alias diagram will be updated until the red button is released. The last position of the cursor before the red button is released is used as the new endpoint for the two alias segments.

B.2.8 "set task parms" mode

This mode is used to specify non-alias initial values for instance variables of static tasks in the model. A task is selected by clicking the red button while the cursor is positioned on the task. A editor for modifying the task name or instance variables will be opened and made the active window.

END

0|6|0|7|8|9

FIN