1

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

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.

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

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.

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

State Machines for System Design

by

Timothy C. Dobbing, Bachelor of Electrical Engineering.

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

Ottawa-Carleton Institute for Electrical Engineering,
Department of Systems and Computer Engineering,
Carleton University,
Ottawa, Ontario, Canada

November 15, 1990

© Timothy C. Dobbing 1990

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-68893-9

Canada

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

State Machines for System Design

submitted by Timothy C. Dobbing, Bachelor of Electrical Engineering.
in partial fulfilment of the requirements for
the degree of Master of Engineering.

Thesis Supervisor

Chairman,
Department of Systems and
Computer Engineering

Ottawa-Carleton Institute for Electrical Engineering,
Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
November 15, 1990

Abstract

This thesis addresses the problems of using StateMate as a tool for Machine Charts design (MachineCharts is a design notation developed by Buhr used in the TimeBench project of which this thesis is a part).

The difficulties in using StateMate as a design tool stem from the fact that it is meant to be used as a tool to specify and analyse the requirements of a system. While it supports the ability to specify temporal and functional aspects, it lacks the ability to specify structural aspects in the style of MachineCharts.

These problems are overcome in this thesis by creating a StateMate model of key MachineCharts components which serve as a set of building blocks upon which a design may be expressed in StateMate terms.

The methods of this thesis provide a way to specify abstract behaviour with StateMate and provide case studies which illustrate the development and application of the model and also provide insight into the use of StateMate when tackling new design problems.

Acknowledgements

I would like to express my thanks to Professor Buhr for his patience, guidance, stimulating conversations and also for the opportunity to work in this research area.

Thanks are also due to Moshe Cohen and Tammy Alfano of i-Logix who enlightened me on the concepts and principles behind StateMate.

Finally, I would like to extend a special note of thanks to Marilyn Sigda, to whom I dedicate this thesis, for the encouragement, support and understanding she gave over the course of this work.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Thesis Objectives	2
1.3	Thesis Contributions	3
1.4	Thesis Outline	3
2	Background	6
2.1	Outline	6
2.2	Introduction	6
2.3	Thesis Approach	10
2.4	MachineCharts	12
2.4.1	Concepts	12
2.4.2	Example	13
2.5	StateMate	14
2.5.1	Concepts	14
2.5.2	Specification Capture	15
2.5.3	Analysis and Verification	22
2.5.4	Code Generation	23
3	Queued, Directed Events in StateMate: A Readers and Writers Case Study	24
3.1	Introduction	24
3.2	Background	25
3.3	Thesis Solution	32
3.3.1	Introduction	32
3.3.2	Resource Manager Activity	35
3.3.3	Queue Request Activity	37
3.3.4	Process Request Activity	41
4	Specification of Abstract Controller Machines (ACMs) with StateMate	45
4.1	Introduction	45

4.2	Active versus Passive State Machines	46
4.3	Impact of AND Notation	48
4.4	Examples	54
4.4.1	Example 1	54
4.4.2	Example 2	56
4.4.3	Example 3	57
5	Generalized Model For Behavioural Specification	59
5.1	Introduction	59
5.2	The Model	60
5.2.1	Functional View	60
5.2.2	Queuing	64
5.2.3	Abstract Controller Machine	69
5.2.4	Engine Behaviour	71
5.3	Analysis and Verification	74
6	Communications Protocol Case Study	76
6.1	Introduction	76
6.2	Functional Description	77
6.2.1	Background	77
6.2.2	Requirements	80
6.3	Design	82
6.3.1	Approach	82
6.3.2	Description	82
6.4	Implementation Example	90
6.4.1	Approach	90
6.4.2	Description	91
6.5	Discussion	99
7	Conclusions	103
7.1	Accomplishments	103
7.2	Future Work	104
	References	106
A	Glossary of Terms	108
B	Implementation examples of Resource Manager	110

List of Figures

2.1	ACM Interactions	8
2.2	Context of this Research	9
2.3	How StateMate is Used With the Thesis Methods	11
2.4	MachineChart Components	12
2.5	MachineChart Example	11
2.6	State Machines	15
2.7	Nesting of States	16
2.8	AND Decomposition	17
2.9	Top_Level_Cntl StateChart	19
2.10	Top_Level ActivityChart	21
3.1	Readers and Writers: System View	25
3.2	State Machines Coupled By Queued, Directed Events	27
3.3	State Machines Coupled By Transient, Broadcast Events	29
3.4	Partitioning of Event Queuing and Event Processing	33
3.5	Thesis Solution for Queuing in StateMate	34
3.6	StateMate Activity Chart of Resource Manager	36
3.7	Queue Request StateChart	38
3.8	Queue Request Simulation Control Program	40
3.9	Process Request (PR_CNTL) StateChart	43
4.1	New PR_CNTL StateChart with Abstract Actions	52
4.2	Resource Manager : Example 1	55
4.3	Resource Manager : Example 2	56
4.4	Resource Manager : Example 3	57
5.1	MachineChart Robot	61
5.2	StateMate Representation of MachineChart Robot	63
5.3	Pushbutton StateChart	66
5.4	Simulation Control Program Template for Queuing	70
5.5	Engine StateChart	72
6.1	Layered Protocol Model	78
6.2	Message Exchange	81

6.3	Link Layer StateChart	86
6.4	Link Layer Connection Component	87
6.5	Link Layer Data Transfer Component	88
6.6	Link Layer Robot	92
6.7	Link Layer Activity Chart	93
6.8	Link Layer Engine StateChart	95
6.9	Link Layer Engine/Pushbutton Interface	96
6.10	Link Layer Pushbutton 1 StateChart	100
6.11	Link Layer Pushbutton 2 StateChart	101
6.12	Multiple Link Layer Connections	102

List of Tables

4.1	Flat Statespace of PR_CNTL StateChart	49
4.2	State/Transition Table for PR_CNTL StateChart	50

Chapter 1

Introduction

1.1 Motivation

The development of reactive systems represents a unique challenge to designers due to the inherent complex patterns of temporal behaviour caused by asynchronous events and concurrency.

The TimeBench [KBWC89] project at Carleton University, of which this thesis is a part, is developing methods and tools for designing reactive systems based around a visual notation developed by Buhr called MachineCharts [Buh90a]. This visual notation combines an extension of Buhr Diagrams [Buh84] used in CAEDE (Carleton Embedded Design Environment) [BIK⁺86] [BK85] [BKHW89] [BKW85a] [BKW⁺85b], an earlier CASE tool also developed at Carleton University, and a state machine notation for specifying temporal behaviour. Together these notations provide a high level view of the system which is both sensible and intuitive and allows the designer to effectively reason about the design and explore alternatives. Analysis tools such as deadlock and critical race detectors will be adopted from the previous CAEDE work and design animation introduced as a means to provide visual feedback to the designer in response to internal and external events.

This thesis takes an alternative approach to the animation of behaviour from that of the main thrust of the TimeBench project by exploring ways of using a commercially available product called StateMate [Har88] to provide the design animation.

StateMate provides a graphical working environment for the specification and analysis of complex reactive systems using Harel's StateCharts notation. The user of the StateMate system is able to prepare, analyse and debug graphical representations of the system from both functional (ActivityCharts) and behavioural (StateCharts) points of view. The tools provided by StateMate allow the dynamic properties of the system to be analysed and also provide a means to simulate the behaviour of the system and watch its execution through animation of its graphical representation [Har88].

The StateCharts notation is of particular interest due to its appeal to practical engineers. It provides a powerful yet simple and intuitive method to capture and visualize temporal behaviour and overcomes problems associated with classical state machine design. These attributes coupled with the savings in development time and cost make it desirable to use StateMate as a design tool.

However, there are difficulties in using StateMate in the design process. StateMate is intended as a tool to specify, debug and analyse the requirements of a system. While it provides the ability to specify temporal and functional aspects, it lacks the ability to specify structural aspects in the way that MachineCharts does. In addition, the underlying event models of the notations used in TimeBench and StateMate differ substantially. The MachineCharts notation assumes a queued, directed event model while the StateMate notations are based on a broadcast, transient event model.

1.2 Thesis Objectives

- The primary objective of this thesis is to provide a method or set of methods which allow StateMate to be used as a tool for design.
- The development and application of these methods should be illustrated through a number of case studies.
- These methods should support the analysis, verification and animation of the design using the tools provided by StateMate.

1.3 Thesis Contributions

- This thesis provides a set of methods which allow StateMate to be used as a tool for design. At the core of these methods is a StateMate model of key MachineCharts components which provides a set of building blocks on which the StateMate design is based. The temporal aspects of the system are specified using StateCharts and the StateMate tools used to analyse, verify and animate the behaviour of the design.

The development of this model led to other equally significant contributions which are listed below.

- This thesis provides a means to specify a queue with StateMate. This was crucial to the development of the StateMate model of MachineCharts components due to the differences in the underlying event models. The flow of information between the components of a MachineCharts design assumes a queued, directed event model whereas information flow between the activities of a StateMate specification is based on a broadcast, transient event model.

The nature of the StateMate event model made the implementation of a queue a non-trivial exercise. StateMate does not provide queuing primitives nor does it provide the data types required to construct a queue. The solution presented overcomes these problems by specifying a queue as a single StateMate activity and through innovative use of the StateMate Analyser and a Simulation Control Program utilizes the host file system to perform the required buffering.

- The case studies of this thesis illustrate the development and application of the model and in doing so also provide a useful tutorial on StateMate which compliments the current documentation supplied with StateMate.

1.4 Thesis Outline

- Chapter 2 provides general background information and then defines the approach of the thesis.

Concepts and terminology used in reactive system design are summarized from [Buh90a] and a brief introduction to the MachineCharts and StateMate notations provided based on [Buh90a], [Buh81], [iL87] and [iL88].

In describing the approach to the thesis, a brief summary of the TimeBench project [KBWC89] as it relates to this thesis is first presented along with the reasons for wanting to use StateMate in the design process. The difficulties in using StateMate for design are then identified and the methods used to overcome them described.

- Chapter 3 addresses issues related to the differences between the event models of StateMate and MachineCharts through a case study of the Readers and Writers problem.

The semantics of each event model are presented based on work done in [BKC90] along with previous attempts to simulate queued, directed events with StateMate.

The weaknesses associated with these attempts stem from the fact that they do not accurately model a queue. This chapter introduces a method to specify a queue with StateMate which overcomes these problems and allows the behaviour of the system to be specified independently of the underlying event model.

The application of this method is illustrated in the solution to the Readers and Writers problem. The solution presented consists of a StateMate activity to queue events and another activity which represents a MachineCharts Abstract Controller Machine to process events.

- Chapter 4 continues with the Readers and Writers case study focusing on the structural and notational issues which impact the specification of Abstract Controller Machines with StateCharts.

These issues are resolved by restricting the types of event, condition and action expressions and their placement within the StateChart.

- Chapter 5 presents the *Generalized Model for Behavioural Specification* which is based on the work done in chapter 3 and chapter 4. It consists of a StateMate

model of key MachineCharts components which serve as a set of building blocks which allow StateMate to be used for design. The behaviour of each component of the model and of the interactions between components is described in detail.

- Chapter 6 illustrates the application of the *Generalized Model for Behavioural Specification* in a case study of a communications protocol.
- Chapter 7 presents the conclusions of the thesis and outlines future work.

Chapter 2

Background

2.1 Outline

This chapter provides the following background material:

- It provides a summary of system design concepts and terminology based on [Buh90a], [BKC90] as they relate to CASE tools and this thesis in particular.
- It illustrates the relation between this thesis and other relevant research (TimeBench) in progress in the Department of Systems Engineering at Carleton University.
- It provides a summary of the issues related to the alignment of the Machine Charts and StateMate notations and the approach taken in this thesis to resolve them.
- It provides information on the MachineCharts and StateMate notations and procedures for describing software systems in sufficient depth to comprehend the case studies presented in the chapters that follow. Complete descriptions may be found in [Buh90b], [Buh90a], [iL87] and [iL88].

2.2 Introduction

Software systems may be described by breaking them down into temporal, functional and structural domains. The temporal aspects of the system refer to the time when

activities take place while the functional aspects refer to the activities themselves. For example, an accounting system may calculate interest for a savings account and add it to the account at the end of the month. The calculation and adding of interest represent functional aspects while the statement "at the end of the month" is a temporal aspect of the system. Structure defines where activities take place and where decisions are made as to when they take place (e.g. accounting department).

In [Buh90b], Buhr assigns these aspects of the system to the development cycle as follows:

- requirements
 - temporal
 - functional
- architectural design of reactive systems
 - temporal
 - structural
 - functional
- implementation
 - temporal
 - functional
 - structural

Functional and temporal aspects of the system are included in each phase of the development cycle while structural aspects are present in the design and implementation phases only. Although temporal information is not commonly included in the design phase of the development cycle, Buhr maintains that it is necessary in order that complex temporal behaviour not be hidden in the source code.

The tools which are employed at each phase of the development cycle may differ substantially. For example, the communication semantics of Ada based systems (rendezvous) differ from those of RMX based systems (message passing). Similarly, the specification of temporal behaviour at the requirements phase may differ from

Figure 2.1: ACM Interactions

that at the design phase due to the differing communication semantics between tools. Examples of different communication semantics include broadcast, transient events (StateMate), instantaneous rendezvous (LOTOS) and queued, directed events (ESTELLE). However, just as an RMX based system can be made to look like an Ada based system through some mapping process, so too can the temporal aspects specified with different tools.

Ideally the temporal behaviour specified at the requirements phase should be directly transferable to the design phase without modification. In this sense the temporal behaviour is simply plugged into the structure of the system.

The concept which embodies the specification of temporal behaviour in this manner is the *Abstract Controller Machine*. Figure 2.1 (adapted from [BKC90]) shows the relationships between ACMs and the *Concrete Machinery* which supports the flow of information between them. ACMs communicate through *Abstract Events*. As in layered systems this communication is not direct but through the layer below. This lower layer is represented by *Concrete Machinery* which is specified during the design phase; ACMs are specified during the requirements phase. The *event/action* interface between ACMs and the supporting *Concrete Machinery* is part of the ACM concept and is fixed. This allows the temporal behaviour encapsulated in the ACM to be plugged into multiple designs providing the the designs are compatible with this interface.

Figure 2.2: Context of this Research

2.3 Thesis Approach

This thesis is part of the TimeBench project [KBWC89] being developed at Carleton University and fits into this effort as shown in figure 2.2.

TimeBench is a CASE tool which emphasizes visual design and evaluation and the generation of high quality implementation code. *Structural* and *temporal* aspects of the design are specified with the *MachineCharts* graphical notation which is described below. *Abstract Controller Machines*, although part of the *MachineCharts* notation, are shown separately to emphasize that they may be specified during the *requirements* phase and *plugged* into the *design* specification at a later date.

StateMate, developed by iLogix, provides a graphical working environment for the specification and analysis of complex reactive systems. *Functional* and *temporal* aspects of the system are captured with the *ActivityCharts* and *StateCharts* graphical editors which are described later in this chapter. Of particular interest is the *StateCharts* notation. It provides a powerful yet simple and intuitive method to specify *temporal behaviour* and as such is well suited to describe *Abstract Controller Machines*. However, there are a number of issues which preclude the direct use of *StateCharts* as a specification tool for ACMs in a *MachineCharts* design. These are:

- Communication Semantics

The communication semantics of *StateMate* and *MachineCharts* differ.

StateMate is based on a broadcast, transient event model while *MachineCharts* assumes a queued, directed event model.

- Behavioural Semantics

The nature of the behaviour defined by a *StateChart* differs from that defined by an *Abstract Controller Machine (ACM)*.

A *StateChart* is associated with a *control activity* in an *ActivityChart* and represents both *temporal* and *functional* aspects of the system. *StateCharts* monitor the behaviour of the surrounding system and influence it directly through interaction with other activities.

Abstract Controller Machines (ACMs), on the other hand, represent *temporal* and *structural* aspects of the system and do not interact directly with the

Figure 2.3: How StateMate is Used With the Thesis Methods

surrounding system.

The *thesis methods* provide a set of procedures which overcome these obstacles and allow the user to specify the *abstract behaviour* encapsulated in an *Abstract Controller Machine* with *StateCharts*. This is illustrated in more detail in figure 2.3. The *thesis methods* add structure to StateMate through the introduction of a StateMate model of a MachineCharts design. This model supports the queued, directed event model used in MachineCharts and serves as a template to be used in the design process. *Abstract Controller Machines* are specified with a StateChart and simply *plugged* into the model.

Referring back to figure 2.2, the *behaviour* of the system may be obtained from either the *requirements simulation* via StateMate or from the *design animation* via *TimeBench* (not complete to the level of StateMate at the time this was written). If both *design animation* and *requirements simulation* are available then the *behaviour* in figure 2.2 represents a checkpoint to verify that the design does indeed reflect the requirements.

Figure 2.4: MachineChart Components

2.4 MachineCharts

2.4.1 Concepts

MachineCharts [Buh90a] is the graphical notation used in TimeBench to describe the structural aspects of the system. The notation is made up of a number of components with meaningful visual representations which are assembled in a sensible and intuitive manner. For example, structural modularity or packaging is captured with a box component while parallelism in the system is specified using a robot component which is visually represented as a parallelogram.

MachineChart components have both an internal and an external view. The external view represents the interface presented to other components of the system while the internal view defines the inner details or workings of the component.

Figure 2.4 shows the subset of the MachineChart components which are used in this thesis and figure 2.5 an example of how the components are assembled to describe the structural aspects of the system.

Referring to figure 2.4, *boxes* and *robots* have already been described. *Work* and *gate buttons* represent two types of interactions. *Work buttons* imply that the *work*

or *activities* associated with the *button* are performed immediately. *Gate buttons*, on the other hand, imply synchronization between two MachineChart components. *Buttons*, both *gate* and *work*, are attached to *robots* and *boxes* respectively to define the interface seen by other components of the system. *Fingers* extend from one component to the *buttons* of another and represent potential interactions between components.

To better understand how these concepts may be applied it is often helpful to consider real life analogies. For example, an automatic teller machine provides an interface consisting of a number of *buttons* which people interact with using their *fingers*. By pushing different *buttons* the automatic teller machine is used (by people) to perform banking functions such as making deposits, withdrawing cash or paying bills.

Engines are primitive components of the MachineCharts notation. Visually they are represented as boxes with an "X" joining the corners indicating that there is no internal view. *Engines* define the activities of the components in which they reside. For example, all *finger* interactions originate from the *engine* of a *robot* [Buh88].

The concepts behind *ACMs* have already been described and are represented by a box enclosing a circular shape implying containment of a state machine.

2.4.2 Example

Figure 2.5 illustrates how these components may be assembled and represents typical MachineCharts diagrams used later in this thesis.

Robot1 is the main focus of the diagram. Its interface is defined by *button1* and *button2* while the internals are defined by the *engine*, *ACM* and *fingers*. *Robot2* and *robot3* have no interface or internal details specified and are considered external system components. This is not to say that they do not have interfaces and internals, they are simply not important and therefore not shown.

Potential interaction between *robot2* and *robot1* is defined by the *finger* extending from *robot2* to the face of *button1*. *Parm1* and *parm2* represent parameters which are exchanged between *robot2* and *robot1* through *button1*.

Since *button1* is a *gate button*, as is *button2*, synchronization is required between *robot2* and *robot1*. *Button1* represents the focal point for the synchronization while

Figure 2.5: MachineChart Example

the *fingers*, extending from *robot2* to the face of the *button* and from the *engine* of *robot1* across the body of the *button* identify the potential for interaction. When the *engines* of both *robots* visit *button1*, then and only then will the work associated with *button1* be performed.

The *finger* extending from *button2* across the body of *button1* indicates that part of the activity associated with *button2*, defined by its own *engine* (not shown, but implied), is the potential to visit *button1* and wait for a visit by *robot2*.

The *ACM* of *robot1* encapsulates its *abstract behaviour*. *Abstract events* are derived from *concrete events* (visits to *button1* and *button2*) and passed to the *ACM* which returns an *abstract action*. This *abstract action* is decoded by the *engine* to determine what activity should be performed next.

2.5 StateMate

2.5.1 Concepts

StateMate allows a user to partition the specification of a system into behavioural and functional aspects with each captured through a separate graphical editor. Be-

Figure 2.6: State Machines

haviour is specified with StateCharts and function with ActivityCharts. StateCharts and ActivityCharts are introduced below with a complete description provided in [1,88].

2.5.2 Specification Capture

StateCharts

StateCharts is the notation used to specify the temporal behaviour of the system in StateMate. It shares many of the same concepts with classical state machines as illustrated in figure 2.6. For example, a state machine is a collection of nodes connected by directed lines termed transitions. Each transition typically has an *event/action* expression associated with it which defines when a transition is taken (occurrence of an event) and what action is performed. StateMate refers to these nodes as OR states and visually represents them as rectangles with rounded corners. The system may be in only one OR state at any particular instant. The classic *event/action* expression is expanded in StateMate to include *conditions* as follows:

- *event[condition]/action*

Upon the occurrence of an *event*, the *condition* is tested and if true, the transition taken and the *action* performed.

Figure 2.7: Nesting of States

The StateCharts notation also allows states to be *nested* and the partitioning of individual states into what are termed *AND components*. The *AND decomposition* and *nesting* of states provides an intuitive and visually pleasing mechanism to specify complex behaviour in StateCharts.

Figure 2.7 (adapted from [iL87]) illustrates the nesting capabilities of StateCharts. The diagram on the left is of a flat StateChart while the diagram on the right represents the equivalent nested StateChart. On the right, the *S* and *T* states have been nested inside a new state *V* with the result that the number of transitions has been reduced. Only one transition from state *V* to state *U*, triggered by the event *F*, is required when nesting is used.

Figure 2.8 (adapted from [iL87]) illustrates the use of *AND decomposition* in StateCharts. Referring to the top diagram, the state *S* is divided into two *AND components*, *T* and *V*, by a dashed line termed an *AND line*. Each *AND component* may be thought of as a visually separate and concurrent StateChart. For example, entering the state *S* causes states *V* and *Y* to be entered in the *T* and *U AND components* respectively. When speaking of state *S* the system is in both the *V* and the *Y* states hence the name *AND components*. The equivalent flat StateChart is shown in the lower portion of figure 2.8. The states *VX*, *VY*, etc represent all the combinations of individual states in the *S* and *T AND components* in the upper part

StateChart with AND Decomposition

Flat StateChart

Figure 2.8: AND Decomposition

of the figure.

Figure 2.9 introduces a typical kind of StateChart used later in this thesis. The *top_level_ctrl* StateChart defines the behaviour of the corresponding *control activity*, described shortly, in an ActivityChart. It consists of two *AND components* which are described below.

- *resource*

The default state is *idle* and upon entry the data item *b_x* is assigned the value zero.

The reception of a *b_start* event causes a transition to the *busy* state and the data item *b_x* to be assigned the value one.

The *busy* state consists of two substates; *process* and *init* (default).

Upon entering the *init* state a transition is made to the *process* state. The event *b_process* is generated and the data item *b_x* decremented by one.

The receipt of additional *b_start* events while in the *process* state causes the data item *b_x* to be incremented by one.

The timeout, represented by the expression *tm(b_process, 5)*, causes a transition out of the *process* state five time periods after the occurrence of the last *b_process* event.

The *condition connector* indicated by the *C* in a circle defines a *decision point* in the StateChart. In this case, a transition is made back to the *idle* state if *b_x* is zero or to the *init* state if *b_x* is greater than zero.

- *status*

This component monitors the status of the resource component.

The default state is *idle* and upon entry the *level_1a* and *level_1b activities* are started. The relationship between this StateChart and these *activities* is explained in the ActivityCharts section later in this chapter.

The occurrence of the event *en(resource.busy)* indicates that the *busy* state of the *resource* component has been entered and causes a transition to the *status.busy* state.

Figure 2.9: Top_Level_Cntl StateChart

Similarly, a transition back to the *status.idle* state is made when the *resource.busy* state is exited.

ActivityCharts

ActivityCharts is the notation used to specify the functional aspects of the system in StateMate. The notation supports the hierarchical decomposition of these aspects into distinct *activities* and the specification of the *information flow* between them. At each level of decomposition a single *control activity* defines a StateChart which controls the behaviour of the other *activities* at that level. These activities are defined by the following states:

- active
- inactive
- hanging

and have the following operations associated with them.

- start
- stop
- suspend
- resume

Control activities manipulate the state of individual *activities* by performing operations on them. For example, the operation *start(activity)* causes the *activity* to become *active*. These operations appear in the corresponding StateChart as *actions* in the *event[condition]/action* expressions which are associated with the *transitions* of the StateChart. In the absence of a *control activity* the *activities* become active at the same time as their parent *activity*.

Information flow between *activities* is of two types; *control* and *data*. *Control* information consists of *events* and *conditions* while *data* information consists of *data items* such as *integers* and *strings*.

Figure 2.10: Top_Level ActivityChart

An example of an ActivityChart is shown in figure 2.10 and illustrates the concepts just described.

Top_Level contains three subactivities; *top_level_cntll*, *level_1a* and *level_1b*. *Top_level_cntll* is a *control activity* whose StateChart is specified in figure 2.9. *Level_1a* on the other hand contains *level_2a* and *level_2b* activities but no *control activity*. The implications of no control activity are summarized as follows:

- The *level_2a* and *level_2b* activities become active at the same time as the *level_1a* activity.
- When the *level_1a* activity becomes *inactive* the *level_2a* and *level_2b* activities also do.
- When the *level_2a* and *level_2b* activities stop (neither one active) then the *level_1a* activity also stops.

In this case the *level_2a* and *level_2b* activities are known as *self terminating activities*.

Control flow in the example is represented by dashed lines. For example, the event *b_start* flows from the *ext_activity* activity to the *top_level_cntll* activity. The response to this event is captured in the *top_level_cntll* StateChart. Data flow is represented by solid lines. *B_Z* characterizes a *data item* flowing between the *level_1a* and *level_1b* activities.

2.5.3 Analysis and Verification

The StateMate Analyser provides a means to analyse the specification of the system and verify its correct operation. It consists of a number of tools briefly described below.

Basic Tests

The Basic Tests tool allows the user to compile the specification to detect syntax errors and check for correctness and completeness.

Dynamic Tests (not used in this thesis)

The Dynamic Tests tool provides capabilities to detect non-determinism and deadlock within the system and to perform reachability analysis.

Simulation

The Simulation tool allows the specification to be executed either interactively or in batch mode. In the interactive mode the user directly manipulates variables, events, etc. while batch mode execution is controlled by a Simulation Control Program written by the user. Status during the simulation may be displayed either graphically or textually.

2.5.4 Code Generation

The StateMate Prototyper provides a Code Generator and a Debugger to aid in the transition from specification to implementation.

- **Code Generator (not used in this thesis)**

The Code Generator provides automatic translation of the specification into compilable Ada code with or without debug facilities. The compiled output from the Code Generator requires linking to a base representing the environment and run time system primitives.

- **Debugger (not used in this thesis)**

The Debugger supports breakpoints, tracepoints etc and may be run in either interactive or batch modes. The histories available from the Debugger reference the specification rather than implementation code.

The StateMate Prototyper thus allows the behaviour of the specification to verified in the target environment and issues such as real time to be addressed at the specification level.

Chapter 3

Queued, Directed Events in StateMate: A Readers and Writers Case Study

3.1 Introduction

The readers and writers problem is chosen for this first case study to introduce the basic concepts of StateMate and MachineCharts and to identify and resolve alignment issues. It is well suited to this purpose due to the inherent simplicity of the problem and because the behavioural characteristics are well known [Buh90a].

The outline of the chapter is as follows:

- Background

The general requirements of the readers and writers problem are outlined followed by a discussion of implementations using different event models based on work done in [BKC90].

- Thesis Solution

The problem of simulating queued, directed events with StateMate is addressed in this thesis through the separation of *abstract behaviour* from *concrete behaviour* and forms the basis for the alignment between the StateMate and MachineCharts notations.

Figure 3.1: Readers and Writers: System View

This partitioning of behaviour is accomplished through the introduction of a StateMate queue activity, representing *concrete behaviour*, which has characteristics similar to those of a MachineCharts gate button. *Abstract behaviour*, which in MachineCharts is encapsulated in an Abstract Controller Machine (ACM), is then able to be specified as a separate StateMate activity independently of the underlying event model upon which StateMate is based.

3.2 Background

The readers and writers problem is essentially one of scheduling access to a resource (e.g. database) shared by a number of processes which are classified as either readers or writers. Processes which read from the database are allowed concurrent access while processes which write to the database require exclusive access. The integrity of the resource is maintained by a resource manager process which schedules access to the database. Figure 3.1 shows the relationship between the *reader*, *writer* and *resource manager* processes of the system and the event flow between them.

- *resource manager*

The *resource manager* provides services to the *readers* and *writers* of the system for access and release of the resource defined by the events *want_r*, *grant_r*, *want_w*, *grant_w* and *fin*. The internal behaviour of the *resource manager* provides the necessary mutual exclusion between *readers* and *writers* and a scheduling function which guarantees fairness.

- *readers*

The *readers* of the system may have concurrent access to the resource. When a *reader* wants access to the resource it generates a *want_r* event to the *resource manager* and waits for a *grant_r* event to be returned before proceeding. The resource is released by generating a *fin* event.

- *writers*

The *writers* of the system require exclusive access to the resource. When a *writer* wants access to the resource it generates a *want_w* event to the *resource manager* and waits for a *grant_w* event to be returned before proceeding. The resource is released by generating a *fin* event.

The underlying communication semantics of the system, or event model, play an important role in the specification of the temporal behaviour for each *reader*, *writer* and *resource manager* process in the system. This is illustrated in figure 3.2 and figure 3.3 which have been adapted from [BKC90] and are described below. In each figure the entities in the system are represented as a set of communicating state machines. Origination of an event is indicated by an exclamation mark (!) preceding the event while receipt of an event is indicated by a question mark (?) preceding the event. Event flow between state machines is represented by the name of the event only and a directed line indicating the direction of event flow.

Referring to figure 3.2, each *reader* is represented by a set of three states: *not_reading*, *waiting* and *reading*. The default state is *not_reading* indicated by the *default transition*. When a *reader* wants to access the resource it generates a *want_r* event (!*want_r*) and makes a transition to the *waiting* state. The *want_r* event flows out of the *reader* state machine across the directed line labeled *want_r* to the *resource_manager*. Assuming the *resource_manager* is in the *idle* state (default), it generates a *grant_r* event and transitions to the *reading* state. The *grant_r* event flows back to the originator of the *want_r* event and causes it (*the reader*) to make a transition to the *reading* state. The occurrence of other *want_r* events (from other readers) while the *resource_manager* is in the *reading* state cause *grant_r* events to be returned with no change in state for the *resource_manager*.

Readers release access to the resource by generating *fin* events and returning to the *not_reading* state. When the *resource_manager* receives these events it remains

Figure 3.2: State Machines Coupled By Queued, Directed Events

in the *reading* state until the last *reader* has released the resource indicated by the expression *?fin[last]* at which point it returns to the *idle* state.

The *resource manager* is able to determine when the last *reader* has released the *resource* by maintaining a count of the number of active *readers*. For every *grant_r* event generated this count is incremented by one. Similarly, for each *fin* event received from a *reader* the count is decremented by one. The last *reader* to release the *resource* causes the count to return to its initial value and a corresponding change in the state of the *resource manager*. This count is not shown in figure 3.2 to keep the diagram simple.

The behaviour of *writers* in the system parallels that of *readers*. However, the treatment of *writers* by the *resource_manager* is different to that of *readers* since *writers* must have exclusive access to the resource. If the *resource_manager* is in the *idle* state when a *want_w* event is received then a *grant_w* event is returned to the *writer* and the *resource_manager* makes a transition to the *writing* state. While in this state subsequent events (except *fin* events) are received and queued for later processing after the active *writer* releases the resource. This includes both *want_r* and *want_w* events. Once the *writer* has released access to the resource, the queued events are processed one by one until the queue is exhausted or another conflict arises over access.

The remaining conflict to be described is the occurrence of a *want_w* event while *readers* are active. In this case the *resource_manager* enters the *finish_reading* state where only *fin* events are processed. Once the last *reader* has released access to the resource a *grant_w* event is generated and a transition made to the *writing* state.

Events in the above example are directed in that the occurrence of an event in the system is tied to a pair of entities (*reader*, *writer* or *resource_manager*) and queued since they may be buffered for later processing should an entity be unable to process an event at the time of its occurrence.

In using StateMate to represent queued, directed events two approaches are possible.

The first is to use global conditions to capture the occurrence of events in the system. While this approach prevents the potential loss of events, the use of conditions to capture events in this manner neither allows events to be queued nor directed

Figure 3.3: State Machines Coupled By Transient, Broadcast Events

and introduces non-determinism through the potential loss of temporal information. For example, if event *E1* occurs at time *T1* and event *E2* occurs at time *T2* the conversion of *E1* and *E2* to conditions *C1* and *C2* causes the temporal information *T1* and *T2* to be lost. The non-determinism occurs at time *T3* (later than *T1* and *T2*) when the system is able to process the conditions but cannot determine the order in which they should be processed.

The second approach is to simulate queued, directed events using StateCharts and also presents difficulties as illustrated in the example below.

The communicating state machine example of figure 3.3 represents similar behaviour to that of figure 3.2 using a broadcast, transient event model. In this example events are broadcast throughout the system instead of being directed towards a single entity. Any entity in the system which is in a state where the occurrence of the broadcast event is valid will detect the event and react to it. Conversely, the transient nature of events dictates that events will be lost if no entities in the system are in a state where the event may be detected.

Referring to figure 3.3, the *reader* and *writer* state machines have the same number of states and event set as those of figure 3.2. However, each instance of a reader requires its own unique set of events to identify the *reader* to the *resource_manager* (e.g. *wr1*, *wr2*, *gr1*, *gr2*, *fr1*, etc). The same would apply for each *writer* if there were more than one.

The directed lines between the *readers* and *writers* in the system and the *resource_manager* identify the flow of events between the various entities (e.g. *wri* = *wr1* or *wr2* etc).

The *resource_manager main* component has the same number of states as the *resource_manager* state machine of figure 3.2 but has unique transitions between states identified for each *reader* and *writer* in the system to support the direction of events. This allows the *resource_manager* to uniquely identify each *reader* or *writer* and process requests on an individual basis. For example, the generation (broadcast) of a *wr1* event by *reader1* will be recognized by the *resource_manager* as originating from *reader1* and result in the broadcast of a corresponding *gr1* event.

Queueing of events by the *resource_manager* is supported through addition of supplementary AND components for each *reader* and *writer* in the system. Fig-

ure 3.3 shows only components for one reader (*rl*) and one writer (*wl*) to keep the diagram simple. Each of these components represents a *queue* where the occurrence of an event is captured until it may be processed by the *main* component. Once the *main* component is able to process the event it is regenerated by the *queue* component and processed as described above.

Although this provides a nice visual representation of what happens in the system it does have some pitfalls.

- It is not general.

The specification for the *resource_manager* must take into account the exact number of *readers* and *writers* in the system.

This is reflected in the number of supplementary AND components required and the number of parallel transitions and events which must be specified in the *main* component.

In addition, as the number of *readers* and *writers* in the system increase the complexity of the specification and the difficulty in testing it will also increase.

- The supplementary AND components do not accurately model a queue.

The use of AND components to model a queue in this manner adds non-determinism to the design. For example, if a *writer* is currently active and two *readers* want to obtain access to the resource the individual *want_r* events will be captured in the appropriate AND components. When the *writer* releases access the two AND components will regenerate the *want_r* events at the same time. The supplementary AND components do not capture the events in a first-in first-out fashion as they would in a proper queue.

- Queuing is imbedded in the behaviour of the Resource Manager.

Queuing is a concrete aspect of a system which is represented in the MachineCharts notation by *gate buttons*. The solution above encapsulates both queuing behaviour and resource management and impacts its usefulness for specifying Abstract Controller Machines (ACMs). The specification for the ACM should exclude the queuing behaviour.

3.3 Thesis Solution

3.3.1 Introduction

The approach taken in this thesis to overcome the problem of simulating queued, directed events with StateMate is through the separation of *abstract behaviour* from *concrete behaviour*.

Referring to figure 3.4, this involves partitioning the *event queuing functionality* of the *resource manager*, represented by the *supplementary AND components*, and the *event processing functionality* of the *resource manager*, represented by the *main AND component* into two separate state machines; the *state machine for queuing of events* and the *state machine for processing of events*.

This partitioning of behaviour is accomplished through the introduction of a StateMate Queue activity, representing *concrete behaviour*, which encapsulates the *state machine for queuing of events* and provides first-in, first-out buffering of requests from *readers* and *writers*. The *state machine for processing of events* represents the *abstract behaviour* of the *resource manager* and is able to be specified independently of the underlying event model upon which StateMate is based.

StateMate Queue Activity

The StateMate Queue activity is to provide first-in, first-out buffering of events. This difficulties in doing this stem from the fact that StateMate is based on a broadcast, transient event model and does not provide queuing primitives nor does it provide a means to specify a queue.

Figure 3.5 illustrates how these problems are overcome through the use of the *Simulation Tool* of the StateMate Analyser to transfer events between the *queue activity* and the *host file system*.

The *Simulation Tool*, as described in the *StateMate Analyser* manual, allows the user to examine the dynamic behaviour of the system.

When used interactively, the user controls the *simulation* by manually triggering *events*, setting *conditions* and changing the value of *data items*.

Alternatively a *Simulation Control Program* can be used to automate the *simulation*. Instead of taking input directly from the user input is read from a file

Figure 3.4: Partitioning of Event Queuing and Event Processing

Figure 3.5: Thesis Solution for Queuing in StateMate

(created by the user) by the *Simulation Control Program* which contains a sequence of *events*, *conditions* or *data items*. It is also possible for the *Simulation Control Program* to create these files itself by collecting *events*, *conditions* and *data items* and storing them for later processing. This is the approach taken in implementing FIFO queues in StateMate.

Referring to figure 3.5, the *Queue StateChart* detects requests to store *items* in the *queue* (*put(item)*) and retrieve *items* from the *queue* (*get(item)*). If the request is to *put an item* in the *queue*, the *item* is copied to a data item which is shared between the *Queue StateChart* and the *Simulation Control Program* and the occurrence of the request captured through a change in state. The *Simulation Control Program* detects this change in state and writes the *item* to *file 1*.

Retrieval of *items* from the *queue* operates in a similar manner. A request to *get an item* from the *queue* is detected by the *Queue StateChart* and captured through a change in state. The *Simulation Control Program* detects this change of state and reads an *item* from *file 2*. This *item* is then copied to a data item shared between the *Queue StateChart* and the *Simulation Control Program* and causes the *Queue StateChart* to return the *item* to the activity which issued the *get(item)* request. If a request to *get an item* from the *queue* is detected while *file 2* is empty and there are *items* available in *file 1*, the *Simulation Control Program* swaps the files and the retrieval of the *item* from the *queue* proceeds as before.

3.3.2 Resource Manager Activity

The functional components of the thesis solution to the Readers and Writers Problem are shown in figure 3.6. The *rm* (*resource manager*) activity is the focus of the ActivityChart and is decomposed into a *qr* (*queue request*) activity and a *pr* (*process request*) activity. The *readers* of the system are depicted by a single external *reader* activity and the *writers* by a single external *writer* activity.

Since *events*, *conditions* and *data items* are global in StateMate, a loosely defined naming convention has been adopted to add clarity to this diagram and to those that follow where the name of an *event*, for example, is prefixed by the name of the *ActivityChart*, *StateChart* or *StateChart component* with which it is associated.

Following this convention, the events for the *rm* activity are *rm_rd* and *rm_wr*

Figure 3.6: StateMate Activity Chart of Resource Manager

which represent requests for *read* and *write* access to the *resource* and *rm_done* which signals the *release* of the *resource*. The *rm_rd* and *rm_wr* events flow to the *qr* activity where they are buffered until such time as the *pr* activity is able to process them. At this point in time, they are transferred from the *qr* activity to the *pr* activity via the data item *rq*. *Readers* and *writers* release the *resource* by generating an *rm_done* event which flows directly to the *pr* activity since this event may be processed at any time.

3.3.3 Queue Request Activity

The *qr* (*queue request*) activity buffers *requests* for *read* and *write* access to the *resource* in a first-in, first-out fashion and delivers them one at a time to the *pr* activity as it is able to process them. The behaviour of the *qr* activity is specified by the *qr_cntl* StateChart of figure 3.7 and accompanying *Simulation Control Program* of figure 3.8.

QR_CNTL StateChart

The *qr_cntl* StateChart of figure 3.7 consists of a *put_q* component and a *get_q* component representing the operations on the *queue*. The behaviour of each component and their interactions is described below.

- *put_q*

The default state is *wt_rq* (*wait request*). Upon entering this state for the first time the data item *pqc* (*put queue count*), representing the number of items in the queue, is initialized to zero. The arrival of an *rm_rd* event or an *rm_wr* event causes the data item *inrqst* to be assigned the value of zero or one respectively and a transition to the *wt_put* state. It is while in this state that the *Simulation Control Program (SCP)* takes over and copies the value of *inrqst* into the input file. The completion of this operation is indicated by the event *rd(inrqst)* and causes *pqc* to be incremented and a transition back to the *wt_rq* state.

- *get_q*

Figure 3.7: Queue Request StateChart

The default state is *wt_rq*. Upon entering this state for the first time the data item *gqc* (*get queue count*), representing the number of items in the queue, is initialized to zero. The arrival of the *rd(rq)* event, representing a *read* operation on the data item *rq*, indicates the ability of the *pr* activity to process a new request. This causes a transition to either the *wt_get* state or the *wt_put* state based on the value of *gqc*.

The *wt_get* state is entered if one or more items are available in the output file. While in this state the *SCP* takes over and retrieves an item from the output file and places it into the data item *outrqst*. This causes the event *wr(outrqst)* to occur, indicating a *write* operation on *outrqst*, and the subsequent transition back to the *wt_rq* state. The value of *outrqst* is copied to *rq* and *gqc* decremented by one.

The *wt_put* state is entered if no items are available in the output file. Again the *SCP* takes over. If input is available in the input file, indicated by the condition $pqc > 0$, then the input and output files are swapped and *pqc* and *gqc* updated appropriately. The condition $gqc > 0$ is then satisfied and a transition made to the *wt_get* state and the retrieval proceeds as described above.

QR_SCP Simulation Control Program

The *qr_scp* Simulation Control Program of figure 3.8 provides the interface between the *qr_cntl* StateChart of figure 3.7 and the host file system. As the *qr_cntl* StateChart captures events to *put items* in the *queue* or *get items* from the *queue* the *qr_scp* Simulation Control Program performs the actual transfer of these *items* between the StateChart and the host file system.

Referring to figure 3.8, the *qr_scp* Simulation Control Program consists of two parts; an *initialization* section which *assigns* logical filenames to physical filenames and a *step-analysis* section which uses *conditions* to detect the current state of the *qr_cntl* StateChart and perform the appropriate file operations. The actions which take place when these *conditions* are met are described below.

- when in (*putq.wt_put*)

```

PROGRAM qr_scp
set trace on;
set go extended;
assign #head 'qr1.dat';
assign #tail output 'qr2.dat';
swap:=0;
STEP_ANALYSIS
  select
    when in(putq.wt_put) =>
      put #tail inrqst;
      rd!(inrqst);
    when in(getq.wt_get) =>
      get #head outrqst;
    when in(getq.wt_put) and pqc>0 =>
      close #head;
      close #tail;
      gqc:=pqc;
      pqc:=0;
      if swap=0 then
        assign #head 'qr2.dat';
        assign #tail output 'qr1.dat';
        swap:=1;
      else
        assign #head 'qr1.dat';
        assign #tail output 'qr2.dat';
        swap:=0;
      end if;
    otherwise =>
      prompt 'Your move...';
  end select;
end STEP_ANALYSIS;

```

Figure 3.8: Queue Request Simulation Control Program

This *condition* indicates that the *qr_cntl* StateChart has detected and captured the occurrence of an *rm_rd* or an *rm_wr* event.

The *put #tail inrqst* statement in the SCP places the value of *inrqst* in the file identified by the logical filename *#tail*.

- *when in (getq.wt_get)*

This *condition* indicates that the *q_cntl* StateChart has received a request from the *pr* activity for the next item in the queue.

The statement *get #head outrqst* transfers the item from the the file identified by the logical filename *#head* to the data item *outrqst* and signals the *qr_cntl* StateChart that the transfer is complete.

- *when in (getq.wt_put) and pqc > 0*

This *condition* occurs when the *#head* and *#tail* files need to be swapped; that is when there is a request from the *pr* activity for the next item in the queue and items exist in the *#tail* file but not in the *#head* file.

The *qr_scp* responds by *closing* both files, *updating* the *pqc* and *gqc* data items and then *assigning* the *#head* and *#tail* logical filenames to the alternate physical files.

This action causes the *qr_cntl* StateChart to make a transition to the appropriate state to retrieve the item from the queue.

- *otherwise*

This *condition* is met when the *qr_cntl* StateChart is waiting for new input, and causes the Simulation Tool to return to the interactive mode where the user is *prompted* for more input (i.e. *Your move ...*).

3.3.4 Process Request Activity

The *pr* (*process request*) activity processes requests for *read* and *write* access to the *resource* in a manner that maintains the integrity of the *resource* and also guarantees fairness between *readers* and *writers*.

The specification of the *pr_cntl* StateChart of figure 3.9 assumes a single *reader* and *writer* in the system and ignores the issue of directing events between the *resource manager* and specific *readers* and *writers*. This is a valid restriction given that the objective of this thesis is to provide a method to specify Abstract Controller Machines (ACMs) with StateCharts. In the MachineCharts environment, of which ACMs are a component, direction of events is provided by *gate buttons*. Direction of events is hidden from ACMs and therefore not required in a StateChart specification of an ACM.

PR_CNTL StateChart

The *pr_cntl* StateChart of figure 3.9 is made up of three components. The *scr* component obtains *requests* for *read* and *write* access to the *resource* from the *qr* activity and *screens* them based on the current status of the *resource* which is reflected by the state of the *rdr* and *wrt* components. The behaviour of each component and their interactions is described below.

- *wrt*

This component represents the *write* status of the *resource*. The default state is *idle* indicating no *writer* active. The reception of the *pr_wrt (write)* event from the *scr* component causes a transition to the *act* state and the generation of a *rm_grant* event. The reception of the *rm_dcnt* event indicates that the *writer* wishes to *release* the *resource* and causes a transition back to the *idle* state.

- *rdr*

This component represents the *read* status of the *resource*. The default state is *idle* indicating no *readers* active. When this state is first entered the data item *rc (reader count)* which indicates the number of active *readers* is set to zero. The reception of the *pr_rd (read)* event from the *scr* component causes a transition to the *act* state, the generation of a *rm_grant* event and *rc* to be incremented by one. More than one *reader* may have access to the *resource* at any given time. Hence, the reception of additional *pr_rd* events while in the *act* state cause the generation of corresponding *rm_grant* events and *rc* to

Figure 3.9: Process Request (PR_CTL) StateChart

be incremented. The reception of *rm_done* event indicates the *release* of the *resource* by one *reader* and causes *rc* to be decremented. When the last *reader releases* the *resource* a transition is made back to the *idle* state otherwise the *act* state is reentered.

- *scr*

This component obtains requests for *read* and *write* access to the *resource* from the *qr* activity and *screens* them based on the status of the *resource* as reflected by the state of the *rdr* and *wrt* components.

The default state is *wt_rq* (*wait request*). When this state is entered a *read* operation is performed on the data item *rq* to obtain a new *request* from the *qr* activity. The occurrence of the event *wr(rq)* indicates the availability of the *request* and causes a transition out of the *wt_rq* state.

If it is a *request* for *write* access to the *resource* (*scr_wr*) and it is *okay to write* (*oktowlr*) then the event *pr_wr* is generated (to the *wrt* component), another *read* operation performed on *rq* and the *wt_rq* state reentered.

If it is not *okay to write* then a transition is made to the *wt_wrt* (*wait write*) state. When the condition *oktowlr* becomes true a transition back to the *wt_rq* state is made and *rd(rq)* and *pr_wr* events generated as before.

Similar logic is followed for *read requests*.

Chapter 4

Specification of Abstract Controller Machines (ACMs) with StateMate

4.1 Introduction

The last chapter focused on alignment issues related to the differences between the underlying event models of StateMate and MachineCharts. These issues were resolved by separating the *event queuing functionality* from the *event processing functionality* of the *resource manager*.

This chapter addresses issues pertaining to the specification of *abstract behaviour* encapsulated within a MachineCharts Abstract Controller Machine (ACM) with StateMate.

The outline of this chapter is as follows:

- Active versus Passive State Machines

Unlike MachineCharts, the distinction between *abstract* and *concrete behaviour* is not inherent in StateMate. The restrictions outlined in this section arise from these differences and dictate which *event*, *condition* and *action expressions* may be used in a StateChart when specifying ACMs.

- Impact of AND Notation

The *pr_ctrl* StateChart which describes the *event processing functionality* of the *resource manager* is examined to determine the impact of the AND notation on the placement of *actions* within a StateChart.

- Examples

This section demonstrates through a number of examples that StateCharts may be used to specify Abstract Controller Machines.

4.2 Active versus Passive State Machines

The Abstract Controller Machine (ACM) is a passive component of the MachineCharts notation whose interface is defined by a set of abstract events and abstract actions. Internally, an ACM is specified by a state machine which detects the occurrence of abstract events and based on the current state returns an abstract action. This abstract action is then translated into one or more concrete actions which are carried out by the MachineCharts component which generated the abstract event.

This is in contrast to the state machine (specified with a StateChart) associated with a StateMate control activity which detects the occurrence of events and performs actions from within the StateChart. In this sense control activities and their associated StateCharts are active components of the system since they may directly influence other components.

Despite this difference in state machine behaviour StateCharts may be made to behave as passive components and therefore be used to specify Abstract Controller Machines. This passive behaviour is achieved by restricting the *event*, *condition* and *action* expressions within the StateChart so that interactions with external components are eliminated. These restrictions are listed below.

- Restrictions for Event Expressions

The following expressions are allowed for events occurring within the ACM:

- *event*
- *changed(variable)*
- *true(condition)*

- *false(condition)*
- *read(variable)*
- *written(variable)*
- *timeout(event, numeric expression)*
- *entered(state)*
- *exited(state)*

The following expressions are not allowed:

- *started(activity)*
- *stopped(activity)*

- Restrictions for Condition Expressions

The following expressions are allowed for conditions occurring within the ACM:

- *in(state)*
- *variable {numeric relation} {numeric expression}*
- *variable {string relation} {string expression}*
- *not condition*
- *condition1 and condition2*
- *condition1 or condition2*

The following expressions are not allowed:

- *active(activity)*
- *hanging(activity)*

- Restrictions for Action Expressions

The following expressions are allowed within the the context of the ACM:

- *event*

- `make_true(condition)`
- `make_false(condition)`
- `variable := expression`
- `read_data(variable)`
- `write_data(variable)`
- `history_clear(state)`
- `deep_clear(state)`

The following expressions are not allowed:

- `start(activity)`
- `stop(activity)`
- `suspend(activity)`
- `resume(activity)`

4.3 Impact of AND Notation

The AND notation used in StateCharts provides an aid to the designer to visualize state machine behaviour. It eliminates the state explosion problem common in classical state machine design from a visual standpoint by allowing the designer to partition the behaviour into logically concurrent components. This apparent concurrency, though, is visual only. Any single external event can translate into a sequence of internal transitions. Internal events may also contribute to the transition sequence.

The *event/action* interface of an ACM represents the *beginning* and *end* of such a transition sequence and dictates the placement of *actions* within a StateChart. The remainder of this section illustrates this by examining the *pr_cntl* StateChart of figure 3.9 in detail as follows:

- The StateChart is translated to a flat representation and the resultant states classified based on position in a transition sequence.

Global State	SCR State	RDR State	WRT State	Type
1	wt_rq	idle	idle	steady
2	wt_wt	idle	idle	transient
3	wt_rd	idle	idle	transient
4	wt_rq	act	idle	steady
5	wt_wt	act	idle	steady
6	wt_rd	act	idle	invalid
7	wt_rq	idle	act	steady
8	wt_wt	idle	act	steady
9	wt_rd	idle	act	steady
10	wt_rq	act	act	invalid
11	wt_wt	act	act	invalid
12	wt_rd	act	act	invalid

Table 4.1: Flat Statespace of PR_CNTL StateChart

- A flat state/transition diagram is presented which represents the desired *abstract behaviour* including a set of *abstract events*.
- A new version of the StateChart is defined which contains these *abstract events* and illustrates their placement within the StateChart.

Referring to figure 4.1, the 12 *global states* on the left are derived from the various combinations of the individual *scr*, *rdr* and *wrt* component states of the *pr_cntl* StateChart of figure 3.9 and are classified by *type* as follows:

- *stable*

Stable states are those in which the state machine is at rest waiting for external stimulus such as request for access to the resource.

These states parallel those of figure 3.2 with *state 1 = idle*, *state 4 = reading*, *state 5 = finish_reading* and *state 7 = writing*. *States 8 and 9* are equivalent to *finish_writing* states but do not need to be explicitly defined in the state machine of figure 3.2 since it is based on a *queued, directed event model*.

Event/State	1	2*	3*	4	5	6	7	8	9	10	11	12
read_rqst	4-1	-	4-1	4-2	-	-	9-3	-	-	-	-	-
write_rqst	7-4	7-4	-	5-5	-	-	8-5	-	-	-	-	-
release_rqst	-	-	-	4(1)-6(7)	5(2)-6(-)	-	1-8	2-	3-	-	-	-

Abstract actions:

1. reader -> active (first one)
2. reader -> active (more than one)
3. reader -> pending release
4. writer -> active
5. writer -> pending release
6. reader -> release; still active
7. reader -> release; resource available
8. writer -> release; resource available

* transient state

Table 4.2: State/Transition Table for PR_CNTL StateChart

● *transient*

Transient states are those which form the intermediate stages of a *transition sequence*. For example, the resource becoming available while there is an outstanding request is a *transient* state since the resource will immediately become busy.

● *invalid*

Invalid states are those which violate the behavioural requirements of the system such as both a reader and a writer being active at the same time.

Abstract actions are placed at the *end* of a transition sequence (i.e. on a transition to a *stable* state) since then and only then will the StateChart be ready to accept new *abstract events*, as a possible result of the *abstract action*.

The state/transition table of table 4.2 represents the desired *abstract behaviour* of the *pr_cntl* StateChart of figure 3.9. Current state (from figure 4.1) is indicated

along the top with *abstract events* down the left side. The first number within the a box is the next state to proceed to while the second number represents the *abstract action* to be returned to the originator of the *abstract event*. A “-” indicates that the state is invalid. States 4 and 5 have two entries for the *release_rqst* event due to the fact that the number of readers is maintained with a count. The figures in brackets represents the *next state/abstract action* information when readers are still active. States 5, 8 and 9 have no *abstract action* associated with the transition since the next state is a *transient* state. *Abstract actions* are only generated on a transition to a *stable* state.

A new version of the *pr_cntl* StateChart is shown in figure 4.1 which has *abstract actions* included as defined in table 4.2. The *abstract actions* have slightly different names due to naming restrictions in StateMate. The cross reference is:

- *rdr_act_frst_one* : reader $\rightarrow$ active (first one)
- *rdr_act* : reader $\rightarrow$ active (more than one)
- *rdr_pndng_rls* : reader $\rightarrow$ pending release
- *wrt_act* : writer $\rightarrow$ active
- *wrt_pndng_rls* : writer $\rightarrow$ pending release
- *rdr_rls_act* : reader $\rightarrow$ release; still active
- *rdr_rls_avail* : reader $\rightarrow$ release; resource available
- *wrt_rls_avail* : writer $\rightarrow$ release; resource available

The following example of two readers accessing the resource followed by a writer illustrate the behaviour captured in table 4.2 and its relationship to table 4.1 and the StateChart of figure 4.1.

The sequence of abstract events is as follows:

- *read_rqst*

The initial state is (*scr.wt_rq*, *rdr.idle*, *wrt.idle*).

Figure 4.1: New PR_CNTL StateChart with Abstract Actions

The reception of a *read_rqst* event causes a transition to the state (*scr.wt_rq*, *rdr.act*, *wrt.idle*) and causes the abstract action *reader* $\rightarrow$ *active (first one)* to be returned.

- *read_rqst*

The arrival of the second *read_rqst* event causes the abstract action *reader* $\rightarrow$ *active (more than one)* to be generated.

The state remains unchanged.

- *write_rqst*

The reception of a *write_rqst* event causes a transition to the state (*scr.wt_wt*, *rdr.act*, *wrt.idle*) and causes the abstract action *writer* $\rightarrow$ *pending release* to be returned.

- *release_rqst*

One of the readers has released the resource causing abstract action *reader* $\rightarrow$ *release; still active* to be returned.

The state remains unchanged.

- *release_rqst*

The last reader has released the resource so the *next state/abstract action* pair in brackets apply.

A transition to the state (*scr.wt_wt*, *rdr.idle*, *wrt.idle*) is made. No abstract action is generated since this is a *transient* state.

Since there is a *write_rqst* pending, a transition to the state (*scr.wt_rq*, *rdr.idle*, *wrt.act*) is made and at this point the abstract action *writer* $\rightarrow$ *active* is returned.

- *release_rqst*

The writer releases the resource causing a transition back to the state (*scr.wt_rq*, *rdr.idle*, *wrt.idle*) and the abstract action *writer* $\rightarrow$ *release; resource available* to be returned.

4.4 Examples

The examples below serve to illustrate the degree of alignment between the MachineCharts and StateMate notations. Each example consists of a different MachineCharts specification of the *resource manager* and accompanying code. The concrete behaviour is represented by the assortment of buttons and different engine agendas while the abstract behaviour is encapsulated in the Abstract Controller Machine (ACM). The abstract behaviour is the same in each example and is specified by a single StateChart.

Appendix B contains the code for the examples which has been generated by hand based on experience with the code generation facility of the ADAGEN CASE tool. The language of implementation is BNR Pascal which is similar to Ada and also parallels the MachineCharts notation in many respects. The ACM code is based on table 4.2. The abstract events (*lgcl_cmt*) and abstract actions (*lgcl_actn*) are listed as part of the interface along with a single function *prc_lgcl_cmt* (*process logical event*). The implementation of this function, although not shown, would be simply a table lookup based on abstract event (*lgcl_cmt*) and current state.

4.4.1 Example 1

The implementation of the resource manager for this first example is shown in figure 4.2. The behaviour represented in the MachineChart parallels that of the StateChart which represents the ACM in that every state in the StateChart may be visited as a result of external events arriving at the robot interface.

The interface provided by the robot consists of two buttons; one to obtain access (*get resource*) to the resource and one to release access (*release resource*). Readers and writers are distinguished from each other when visiting the *get resource* button by a parameter indicating the type of access required (*read, write*). When readers and writers visit the *get resource* button their request for access to the resource is converted into an *abstract event* and passed to the ACM. The *abstract action* returned determines the next action to be performed. The visit with the reader or writer may be either completed or, in the case of a conflict, the *release resource* button visited to wait until the resource is free. Once the resource becomes available

Figure 4.2: Resource Manager : Example 1

Figure 4.3: Resource Manager : Example 2

the visit is completed and access to the resource granted to the waiting reader or writer.

4.4.2 Example 2

This example depicted by the MachineChart of figure 4.3 differs from the first in a number of ways. *Read* and *write* access is provided by separate buttons which has scheduling implications regarding fairness and starvation for both Ada and BNR Pascal. In this example, only the *get write access* button visits the *release resource* button when a conflict is detected while subsequent *readers* are blocked whenever a *writer* has access to the resource. This information (*readers blocked*) is not available

Figure 4.4: Resource Manager : Example 3

to the ACM and as such not all states may be visited as in the previous example. Behaviour of the *resource manager* is thus distributed between the ACM and the robot implementation.

4.4.3 Example 3

The MachineChart of figure 4.4 differs from the previous two in that the ACM is not aware of conflicts between *readers* and *writers*. The robot implementation differs substantially from the ACM specification and the implementation code in Appendix B reflects the complexity.

The interface provided by the robot consists of four buttons; *get read access*, *get*

write access, *wait write access* and *release resource*. Readers obtain access to the resource by visits to the *get read access* button and are blocked whenever a writer has access to the resource or is waiting to access the resource. Writers access the resource by making visits to the *get write access* and *wait write access* buttons. If the resource is available a writer is only required to visit the *get write access* button. However, if the resource is not free the writer receives a *wait* indication (instead of a *grant* indication) at the completion of the visit and then visits the *wait write access* button to wait until the resource is available. All buttons are blocked except the *release resource* button until the resource becomes available at which time the waiting *writer* is allowed access. The *get read access* and *get write access* buttons are blocked whenever a *writer* is active.

Chapter 5

Generalized Model For Behavioural Specification

5.1 Introduction

The Readers and Writers Case Study of chapter 3 and chapter 4 provides a means to specify the behaviour of system components independent of their structure. This allows the behaviour of the system to be partitioned between StateMate and MachineCharts such that the abstract behaviour, specified with StateMate, is easily integrated with the structural description, specified with MachineCharts.

Another significant result of the case study deals with information flow between system components. StateMate and MachineCharts differ in the manner in which events and data are exchanged between system components in that MachineCharts inherently provide a mechanism to queue such information while StateMate does not. This was a major obstacle which had to be overcome before any sort of alignment was possible between the two notations. The Readers and Writers Case Study provides a means to do this by specifying a queue as a separate StateMate activity and through the use of the StateMate Analyser and a Simulation Control Program utilized the host file system to perform the required buffering.

Although these are significant achievements the model is specific to the Readers and Writers solution has the following limitations.

- An arbitrary number of request queues is not supported.

Only one queue for requests is implemented and the interaction between it and the Abstract Controller Machine (ACM) is tightly coupled.

- External requests are not treated in a uniform manner.

Requests for resource access flow through the queue activity while requests to release the resource bypass the queue and are interpreted directly by the ACM.

This chapter provides a generalized model which overcomes the above restrictions and serves as a template for behaviour specification.

5.2 The Model

5.2.1 Functional View

The MachineChart and StateMate notations are quite different in their approach to system specification with respect to the partitioning of behaviour and function and to the flow of information between the components which make up the system. The MachineCharts notation contains a number of well defined functional components such as engines and pushbuttons which serve as building blocks on which the final system specification is based. For example, pushbuttons (and fingers) provide an interaction function while engine provide an agenda function. StateMate, on the other hand, does not have such a predefined set of building blocks. It is up to the user to define these using the Activity and StateChart editors.

The approach taken here is to create a set of StateMate building blocks which is equivalent to the various functional components of the MachineCharts notation. The representation of the behavioural aspects of the MachineCharts notation with StateMate forms the model and provides a vehicle to integrate the two notations allowing the structure to be specified with MachineCharts and the abstract behaviour to be specified with StateMate.

Figure 5.1 shows a typical MachineChart robot consisting of three robots: *robot_1*, *robot_2* and *robot_3*. *Robot_2* and *robot_3* interact with *robot_1* through the pushbutton interface defined by *pushbutton_1*, *pushbutton_2* and *pushbutton_3*.

Figure 5.1: MachineChart Robot

The *engine* interprets these interactions and generates an appropriate *abstract_event* to the *ACM* (*Abstract Controller Machine*) which returns a corresponding action (*actn*).

Figure 5.2 shows a StateMate representation of the MachineChart diagram of figure 5.1. Each robot is represented as a StateMate activity. *Robot_1* contains a number of subactivities defining the pushbuttons, engine and abstract controller machine while *robot_2* and *robot_3* are depicted as external activities. The interactions between *robot_2* and *robot_3* and the pushbutton activities of *robot_1* are identified by the events *concrete_event_1*, *concrete_event_2* and *concrete_event_3*. The pushbutton activities receive these events and buffer them until the engine activity is ready to process them. The concrete events, through the process of buffering, are delivered to the engine activity as *data1*, *data2* and *data3*. The *engine*, as mentioned above, interprets the incoming data (representing the concrete events) and generates an appropriate *abstract_event* to the *ACM* which returns a corresponding action.

This more general model differs from that used in the Readers and Writers case study with the addition of an engine activity which serves as a buffer, so to speak, between the abstract behaviour of the robot, represented by the *ACM*, and the structure of the surrounding system, represented by the pushbuttons.

Differences in the Two Representations

Although the StateMate model of the MachineChart robot of figure 5.1 is sufficient to allow the specification of an Abstract Controller Machine using StateCharts there are a number of differences between the MachineChart and StateMate representations worth noting.

- Concrete events while inherent in the MachineCharts notation must be explicitly defined with StateMate.
- Information flow through MachineChart pushbuttons is bidirectional whereas it is only one way with the StateMate representation. The reply information, if required, must be generated by the engine activity.

Figure 5.2: StateMate Representation of MachineChart Robot

5.2.2 Queueing

There are a number of situations where queueing is required within a system. These requirements exist both at the application level and at the run time system level. Two possible situations where queueing is required are outlined below.

- **Flow Control**

In a data communications system traffic flow is bursty in nature. Even though the capacity of the system is not exceeded on average there may be times when the amount of information entering the system exceeds its ability to process it. The new information must be queued until such time that the system is able to process it.

- **Mutual Exclusion**

The Readers and Writers case study of Chapter 3 is an example where queueing is required to allow requests to be buffered until such time (driven by some condition within the system) that they may be safely processed.

Unlike the first example, the StateChart may be able to accept the request however it is invalid at this point in time (defined by system state) and processing of the request must be delayed.

As mentioned above, StateMate does not provide a mechanism to buffer information flowing between activities. If an event is generated but no activity is ready (waiting) to receive it then it is lost. This applies to information flows of all kind, events, data and conditions.

Queueing is handled by the pushbutton activities of the model. The required elements to implement queueing are:

- **Pushbutton Activity**

The pushbutton activity represents the queueing function within the system.

- **Pushbutton Control Activity and StateChart**

The control activity and associated statechart specify the behaviour of the pushbutton activity. The queue is implemented using a double buffer approach

consisting of an input buffer and an output buffer to allow concurrent access. The buffers are swapped should the output buffer become exhausted when input is available from the input buffer.

- Simulation Control Program (SCP)

StateMate does not provide a means to specify complex data structure such as queues, lists, arrays etc. Only simple data types are supported including bytes and integers.

A Simulation Control Program (SCP) is required to allow the host file system to be utilized as a queue. The Simulation Control Language (SCL) provides primitives to detect events, conditions, changes in data items, state information and of course the ability to manipulate files (open ,close, read and write).

The SCP is run under the StateMate Analyser along with the pushbutton StateChart providing an interface between the StateChart and the host file system.

The statechart for *pushbutton_1* is shown in figure 5.3 and consists of the following elements:

- *put_q* and *get_q* AND components

These represent the storage and retrieval operations for the queue. The *put_q* component receives an event and converts it to an integer value for storage in the buffer. The *get_q* component receives an event to retrieve an item from the queue and performs the necessary actions which may involve swapping the input and output buffers or waiting until input is available.

- *pqc, qqc*

These are state variables which indicate the number of items in the input and output buffers respectively.

- *inrqst, outrqst*

These are shared variables used by the StateChart and the Simulation Control Program.

Figure 5.3: Pushbutton StateChart

- External Events

External events may originate from either the surrounding system *cel* (*create_event_1*) or from the robot engine *rd(data1)*.

A state by state walkthrough of the *pushbutton* behaviour is described below. States are uniquely identified by concatenating the *AND* component name with the local state name separated by a period.

- *put_q.wt_rq*

This is the default state for the *put_q* component. Upon entering this state the state variable *pqc* (*put queue count*) is initialized to zero. The receipt of a *cel* event causes a zero to be written to *inrqst* and a transition to the *put_q.wt_put* state.

- *put_q.wt_put*

It is during this state that the data, *inrqst*, is stored in the input buffer. This state differs from the *put_q.wt_rq* state in that actions are being performed while actually in the state rather than on a transition to the next state. Upon completion of the actions (write to buffer) the state is exited and a transition made back to the *put_q.wt_rq* state. During this transition the state variable, *pqc*, is incremented to indicate a new item has been placed in the input buffer.

- *get_q.wt_rq*

This is the default state for the *get_q* component. Upon entering this state the state variable *gqc* (*get queue count*) is initialized to zero. The receipt of a *rd(data1)* event indicates a request to retrieve an item from the queue. If data is available in the output buffer, indicated by *gqc* then the *get_q.wt_get* state is entered otherwise the *get_q.wt_put* state is entered. No actions take place during these transitions.

- *get_q.wt_get*

This state is entered due to an external request to retrieve data from the output buffer. It may be entered directly or after input and output buffers

have been swapped (see *get_q.ut_put* state description). This state, as with the *put_q.ut_put* state, has actions associated with it which retrieve an item from the output buffer. These actions begin execution upon entry to the state and when completed cause a transition back to the *get_q.ut_rq* state. During this transition the data removed from the queue is delivered to the requestor and the state variable, *gqc* decremented to indicate removal of an item from the output buffer.

- *get_q.ut_put*

This state is entered due to an external request to retrieve data from the output buffer when there is no data available indicated by the state variable *gqc*. This state has a number of actions associated with it which begin execution upon entry. The first action involves monitoring the state of the input buffer to test for the presence of data. The pushbutton remains in this state until data becomes available triggered by a *cdt* event which will cause the *put_q* component to execute. When data is available the input and output buffers are swapped, the state variables *pqc* and *gqc* updated and a transition made to the *get_q.ut_get* state which will perform the actual retrieval of data from the output buffer as described above.

Two possibilities exist to specify the actions associated with the various states identified above.

- Static Reactions

The Static Reactions feature of StateMate provides a means to associate a series of actions with entry into a state.

This method could be used for the *put_q.ut_put* and *get_q.ut_get* states but is not sufficient for the *get_q.ut_put* state since it must continuously monitor the status of the input buffer to detect the presence of data. Also since StateMate V2.5 does not support the complex data structures required to specify the input and output buffers and provides no mechanism to perform file manipulation directly the Static Reactions feature is not suitable.

- Simulation Control Program (SCP)

A Simulation Control Program executes along with the StateMate specification under control of the StateMate Analyser.

The Simulation Control Language (SCL) used to write an SCP provides a number of primitives to interact with both the StateMate specification and the host file system.

This method overcomes the restrictions outlined above allowing files to be used as input and output buffers and providing a means to continuously monitor the status of the input buffer from the *get_q.wt_put* state and perform the input/output buffer swap at the appropriate time.

Figure 5.4 contains a single Simulation Control Program (SCP) template which is used to manage the buffering for an arbitrary number of pushbutton activities. As mentioned above, a double buffering scheme is used consisting of two files per pushbutton; an input file for storing buffer items an output file for retrieval of buffer items. When the output file is exhausted and input is available, the two files are swapped and processing continues.

The SCP consists of two sections; an initialization section which performs the initial assignment of the input and output files for each pushbutton activity and the body which reacts based on the state of the individual pushbuttons to perform read, write and swap operations as required.

Although the behaviour of each pushbutton is the same as another, the information to be buffered may vary. This is accommodated by simply defining data items specific to each pushbutton. If more than one piece of information is associated with a pushbutton the SCP performs a number of read or write operations on the appropriate file.

5.2.3 Abstract Controller Machine

The Abstract Controller Machine (ACM) specifies the behaviour of a MachineChart robot dealing purely with events and actions as opposed to the concrete behaviour which is tied to the actual robot structure (number of pushbuttons etc).

```

program general_scp;
  assign #head_n 'qr1_n';
  assign #tail_n output 'qr2_n';
  swap_n := 0;
  .
  .
  above statements duplicated for each additional pushbutton.
STEP_ANALYSIS
select
  when in (putq_n.wt_put) =>
    -- store data (inrqst_n) in the input file
    put #tail_n inrqst_n;
    rd!(inrqst_n);
  when in (getq_n.wt_get) =>
    -- retrieve data from the output file
    get #head_n outrqst_n;
  when in (getq_n.wt_put) and pqc_n > 0 =>
    -- close both files, swap them and update state variables
    close #head_n; close #tail_n;
    gqc:=pqc; pqc:=0;
    if swap_n=0 then
      assign #head_n 'qr2_n'; assign #tail_n output 'qr1_n';
      swap_n:=1;
    else
      assign #head_n 'qr1_n'; assign #tail_n output 'qr2_n';
      swap_n:=0;
    endif;
  .
  .
  above statements duplicated for each pushbutton
END select;
END STEP_ANALYSIS;

```

Figure 5.4: Simulation Control Program Template for Queuing

This distinction means that the specification of the ACM may be developed independently of the structural specification and as a result is portable, meaning that it is applicable to any structural specification. The structural specification can thus be optimized to a particular environment without affecting the overall system behaviour.

Since the ACM is purely application dependent, no template is provided however certain restrictions do apply regarding the specification of expressions within the ACM. These restrictions are necessary to ensure that the specification of the ACM is indeed completely removed from that of the surrounding system. In general, except for the interface shared with the engine activity (*abstract_event* and *actn*), the ACM is restricted from knowing anything about its environment. These restrictions have been detailed in Chapter 4.

5.2.4 Engine Behaviour

The *engine* activity of figure 5.2 coordinates the interaction between the *ACM* activity and the *pushbutton* activities.

A template for engine behaviour specification is shown in figure 5.5 and consists of the following elements:

- *pb1, pb2, pb3, acm*, and *engine*

These identify the pushbutton interface components, Abstract Controller Machine (ACM) interface component and core engine component.

The interface components interact directly with the external activities to facilitate information transfer between these activities and the engine activity while the core component controls the behaviour of the engine activity as a whole.

- Internal Events

pb1_enable, *pb2_enable* and *pb3_enable* are used by the *engine* component to control the pushbutton interface components.

- Internal Conditions

Figure 5.5: Engine StateChart

The *engine* component reacts to external events based on the state of the *acm* and pushbutton (*pb1*, etc) interface components through the conditions *in(pb1.wait)*, etc.

- External Interface

The *engine* activity interacts with the *pushbutton* and *acm* activities through a number of variables; *data1*, *data2*, *data3*, *abstract_event* and *actn*.

A state by state walkthrough of the *engine* StateChart to illustrate its behaviour is described below.

- *pb1.wait*

This is the default state for a pushbutton interface component. Upon receipt of a *pb1_enable* event a transition is made to the *pb1_accept* state and the event *rd(data1)* generated indicating to the pushbutton activity (*pushbutton_1* in this case) that the *engine* activity is ready to process event input.

- *pb1.accept*

This state indicates that the *engine* activity has requested input from the associated pushbutton activity. The receipt of the event *wr(data1)* indicates that input is available and causes a transition to the *pb1.wait* state.

- *engine.wt_pb*

This is the default state for the core engine component. Upon initialization of the activity the events *pb1_enable*, *pb2_enable* and *pb3_enable* are generated to indicate to the pushbutton activities that the engine activity is ready for external input. This action is, of course, application dependent. The application may dictate that only one pushbutton be enabled, for example.

The *condition* connector (c) provides a branch operator and allows *conditions* to have unique actions associated with them. The branch for the *pb1* interface component is shown.

The *in(pb1.wait)* condition becomes true when the *pb1.wait* state is entered and causes a transition from the *engine.wt_pb* state to the *engine.wt_acm* state.

During the transition the variable *abstract_event* is assigned the value 1 (arbitrary value for the example) which generates the event *wr(abstract_event)*.

- *engine.wt_acm*

This state is entered after an event has been generated to the ACM activity and a response is expected. The condition *in(acm.wait)* causes a transition back to the *engine.wt_pb* state. During the transition the action *pre_action* is executed which is dependent on the specific application.

- *acm.wait*

This is the default state for the *acm* component of the StateChart. The receipt of the event *wr(abstract_event)* causes a transition to the *acm.accept* state. During the transition the event *rd(actn)* is generated which indicates to the ACM activity that the engine activity is waiting for a response.

- *acm.accept*

This state is entered after an *abstract_event* has been sent the ACM activity. The receipt of the event *wr(actn)* indicates that the ACM activity has replied and causes a transition to the *acm.wait* state.

While the ACM statechart specification is independent of the system structure the engine statechart details are not. The number of pushbuttons required will vary on the implementation and although the structure of each remains the same, ignoring data considerations, the core engine component will vary since it must interact with the surrounding system and perform the translation between the concrete and abstract aspects of the system.

5.3 Analysis and Verification

Since each component of a MachineChart robot is represented by a StateChart activity each may be debugged and analyzed separately or in combination with each other using the tools provided by StateMate. The tools provide basic tests for errors and completeness and dynamic tests for race conditions and indeterminism. Also, by using the Simulation Control Program language available as part of

the StateMate Analyzer, which was used to implement queues, different scenarios may be constructed and repeated for different implementations to verify correct behaviour of the system.

Chapter 6

Communications Protocol Case Study

6.1 Introduction

The layered communications protocol presented in this case study is used to demonstrate the application of the *Generalized Model for Behavioural Specification* outlined in Chapter 5.

The outline of this chapter is as follows:

- Requirements

The requirements of the *link layer* are presented based on ISO layered communication models described in [Sta85] and message exchange sequences described in [Hou79].

- Design

The approach for the design is to specify the Abstract Controller Machine based on the functional requirements without regards for system structure. The system events associated with *link layer* behaviour are identified and mapped to the *Abstract Event/Abstract Action* interface between the ACM and Engine. The StateChart for the ACM is then presented and the various aspects described.

- Implementation

An implementation example is presented illustrating how the StateChart which describes the ACM may be used in different structural environments.

- Discussion

The discussion examines the ACM interface and how it may be utilized to simplify the specification of the associated StateChart and how this enables the ACM to be applicable to a broader range of problems.

Making full use of the *Abstract Event/Abstract Action* interface allows the ACM to support a flexible window size, for example, and also allows the single ACM specification for the *link layer* to be used in an application supporting many link layer connections.

6.2 Functional Description

6.2.1 Background

Figure 6.1 shows the typical layered protocol model consisting of two nodes made up of a *user*, *link layer*, *physical layer* and a *transmission medium* between them. Each layer of the protocol stack provides a set of service primitives to its higher layer and interacts with its associated layer on the remote node using the services provided by the lower layer. The services are implemented through a peer to peer protocol.

The *transmission medium* represents the physical medium across which the information flows between *users*. The medium is characterized by errors which may impair the transmission of information causing the information to be received incorrectly or to be lost all together.

The *physical layer* provides a means to convert information supplied by the higher layer into a suitable format for transmission. This involves converting the user information into waveforms which can be efficiently transmitted over the *transmission medium*. As mentioned above, this transfer of information is subject to the effects of noise etc. which may cause it to be lost or delivered incorrectly.

Figure 6.1: Layered Protocol Model

The purpose of the *link layer* is to overcome these problems by providing reliable information transfer between nodes. Typical *link layer* functions include:

- link management

Procedures are provided which allow the establishment, termination and maintenance of connections used for information transfer between nodes.

- data transfer

Procedures are provided to allow the users of the *link layer* to exchange information once a connection is established. The unit of information transfer is known as a frame.

- error detection and correction

The *link layer* provides reliable information transfer between nodes.

Frames sent between nodes may be lost or received incorrectly due the nature of the *transmission medium*. To provide reliable information transfer errors must be detected and corrected.

Detection of errors is accomplished by appending a cyclic redundancy check (CRC) to the end of a frame before it is sent across the data link connection. When the frame is received, the CRC is recalculated and compared to the received CRC. If they differ the frame is discarded.

Correction of errors, once detected, is achieved by many possible methods. Two common methods are *Stop-and-Wait ARQ* (*Automatic Repeat Request*) and *Go-Back-N ARQ*. The concept behind both these methods is that as frames are received correctly, an acknowledgement is sent back to the sender. If an error is detected then a negative acknowledgement is sent. Since these special frames may also be corrupted timers are employed by the sender. If no reply is received from the destination node after some predetermined time then the original frame is retransmitted. Frames are numbered to uniquely identify them allowing duplicate frames to be detected and processed accordingly. The *Stop-and-Wait* protocol sends one frame and waits for an acknowledgement while the *Go-Back-N* protocol allows a number of frames to be sent before an acknowledgement is required.

The *user* layer is at the top of the protocol stack and represents the application running on the node. It uses the services of the *link layer* to establish connections with the remote node and exchange information without concern for the problems associated with the physical transmission.

6.2.2 Requirements

This case study is interested in modelling the *link layer*. The services provided by the *physical layer* will not be examined. It appears to the *link layer* entities that they are communicating directly with each other just as the *users* think they are communicating directly with each other.

The services provided by the *link layer* to the *user* layer are defined by the following set of primitives.

- dl_connect.request
- dl_connect.indication
- dl_connect.response
- dl_connect.confirm
- dl_disconnect.request
- dl_disconnect.indication
- dl_data.request
- dl_data.indication

The messages exchanged between peer *link layer* entities are:

- dl_open
- dl_close
- dl_data
- dl_data_ack

Figure 6.2: Message Exchange

Reliable transmission is accomplished using a variation of the *Stop-and-Wait* protocol described above using timers at the source and numbering of frames. Negative acknowledgements are not used in order to simplify the example.

A typical message exchange sequence is shown in figure 6.2. The *user* on the left initiates a connection using the *dl_connect* primitive and exchanges some information with its peer using the *dl_data* primitive. The *dl_disconnect* primitive is used to terminate the connection.

6.3 Design

6.3.1 Approach

The approach taken to design the *link layer* entity is to specify the abstract behaviour considering only the event interfaces between *user* and *link layer* entities and between peer *link layer* entities. The *physical layer* is ignored with the assumption that peer *link layer* entities communicate directly with each other. This is not unreasonable since the nature of layered protocols implies peer to peer communication plus service to the next higher layer.

6.3.2 Description

Event Definition

The following events flow from the *user* layer to the *link layer*:

- connect_request
- connect_response
- disconnect_request
- data_request

The following events flow from the *link layer* to the *user* layer:

- connect_indication

- *connect_confirm*
- *disconnect_indication*
- *data_indication*

The following events flow between the *link layer* entities.

- *open*
- *close*
- *data*
- *ack*

Input to the ACM is mapped into a set of *Abstract Events*. This input may be in the form of concrete events, conditions and data or combinations of all three. Similarly, output from the ACM denoted by *Abstract Actions* is mapped into concrete events, conditions and data or combinations thereof.

The *Abstract Events* and *Abstract Actions* which define the interface between the ACM and the Engine are detailed below with explanations where appropriate.

Abstract Events

The following input is defined for the ACM:

- *connect_request*
- *connect_response*
- *disconn_request*
- *data_request*
- *ll_open*
- *dl_close*

- *dl_data_0, dl_data_1*

Two separate events are required to uniquely identify the numbering of data frames.

- *dl_ack_0, dl_ack_1*

Similarly, two events are also required to identify the associated acknowledgement frames.

Abstract Actions

The following output is defined from the ACM:

- *issuc_cc*

Issue a connect_confirm.

- *issuc_ci*

Issue a connect_indication.

- *issuc_di*

Issue a disconnect_indication.

- *send_a0*

Send acknowledgement with sequence number of zero.

- *send_a1*

Send acknowledgement with sequence number of one.

- *issuc_di_send_a0*

Issue a data_indication and send acknowledgement with sequence number of zero.

- *issuc_di_send_a1*

Issue a data_indication and send acknowledgement with sequence number of one.

- *send_d0*

Send frame with sequence number of zero.

- *send_d1*

Send frame with sequence number of one.

- *send_open*

- *send_close*

- *ready_new_data*

Indicate that the link is available for transfer of new information.

- *disconn_confirm*

Indicates that the remote entity has responded to the *dl_close* previously sent.

StateChart

A StateChart describing the behaviour of the link layer entity is shown in figure 6.3 with the *connect* and *data_transfer* components shown in figure 6.4 and figure 6.5 respectively.

A Simulation Control Program (SCP) is used to simplify the visual appearance of the StateChart, specifically at the Engine/ACM Interface defined by the *input* component.

The StateChart consists of the following components:

- *input*

This component receives input from the Engine in the form of *Abstract Events* and generates the appropriate internal events.

StateMate V2.5 does not support enumerated types. External events from the engine are mapped into an integer value (*abstract_event*) before transfer to the ACM. The ACM receives this *abstract_event* and converts it from its integer representation back to events.

Figure 6.3: Link Layer StateChart

A resolution test chart featuring various patterns of vertical and horizontal lines. Each pattern is accompanied by a numerical value representing its resolution. The values include 1.0, 1.1, 1.25, 1.4, 1.6, 1.8, 2.0, 2.2, 2.5, 2.8, 3.2, 3.6, 4.0, 4.5, 5.0, 5.6, 6.3, 7.1, and 8.0. The patterns become progressively smaller and more densely packed as the numerical value increases.

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

Figure 6.4: Link Layer Connection Component

Figure 6.5: Link Layer Data Transfer Component

The condition *tr(scp_dl_input)* causes the Simulation Control Program (SCP) to run, transforming the input (*abstract_event*) from an integer value to a unique StateMate event. These events drive the remainder of the StateChart.

- *output*

This component performs a similar function for information flow out of the ACM.

The default state is the *busy* state indicating that the engine is not prepared to process output defined by the data item *abstract_action*.

Two additional states, *wt_action* and *wt_read*, are used to avoid race conditions between the engine and the ACM.

The *wt_read* state is entered should the ACM complete its processing indicated by a write to *dl_action*. The event *rd(abstract_action)* causes *dl_action* to be copied to *abstract_action* and a transition back to the *busy* state.

The event *rd(abstract_action)* occurring in the *busy* state causes a transition to the *wt_action* state. This state indicates that the engine is ready for the *abstract_action* but the ACM is still processing the *abstract_event*. When the ACM has completed its processing the write to *dl_action* causes it to be copied to *abstract_action* and a transition back to the *busy* state.

- *connect*

The *connect* component manages the connection associated with the *link layer* entity. The default state is *closed*. Transitions to the *open* state may be made due to requests by the local node, indicated by the intermediate state *connect_out*, or by requests from the remote node, indicated by the *connect_in* state.

- *data_transfer*

The *data_transfer* component consists of a number of states which are in turn divided into separate *and* components and substates.

The default state is the *idle* state. A transition to the *busy* state is caused by entry into the *connect.open* state.

The *data_transfer.busy* state is broken down into two *and* components representing the transmission (*tx*) and reception (*rx*) of information between peer link layer entities.

Each of these components contain two states, *wt_0* and *wt_1*, which represent the numbering of frames with sequence numbers (0,1) to uniquely identify them.

These states are further subdivided, for the *tx* component, into *idle* and *busy* states. The *idle* state indicates that the data link layer connection is idle (no information transfer in progress) and that the *user* layer may request the transmission of information through the *data.request* service primitive. The *data_request* event indicates data to be transferred and the *busy* state is entered. The *dl_timeout* event causes the information to be retransmitted over the data link layer connection while the reception of a *dl_ack_0* event will cause a transition to the *wt_1* state.

Should the *connect.opc.1* state be exited at any time or in any state of the *data_transfer.busy* state, a transition from the *data_transfer.busy* state to the *data_transfer.idle* state will be made.

6.4 Implementation Example

6.4.1 Approach

The approach taken for this example is to apply the *Generalized Model For Behavioural Specification* developed in Chapter 4 to model a MachineChart specification of the *Link Layer* with StateMate.

The focus is on the engine and pushbutton activities which together define the structural behaviour of the *Link Layer* implementation. The ACM StateChart describing the abstract behaviour of the *Link Layer* has been developed above independent of the structural implementation and is used to drive the structural behaviour. This example thus illustrates the smooth integration of abstract behaviour with structural implementation.

Simulation Control Program

A Simulation Control Program (SCP) is used to implement functions which are not supported by StateMate V2.5. These include enumerated event types and complex action definitions.

Restrictions/Limitations

The example given is operationally incomplete. Only enough information is included to illustrate the procedure developed in Chapter 4. Deadlock issues, for example, are ignored.

No attempt is made to detect or recover from departures from the protocol; either peer to peer or interlayer.

6.4.2 Description

MachineChart

Figure 6.6 represents one possible implementation of the communications subsystem with MachineCharts. It consists of three robots representing the *user layer* (*ul*), *physical layer* (*pl*) and *link layer* sometimes called the *data link layer* (*dl*). Since this case study is concerned primarily with the *data link layer*, only details pertaining to it are examined. The issues of deadlock related to robot communication are ignored since they have no bearing on the internal interactions of the *data link layer* entity.

The *dl* robot provides an interface consisting of two pushbuttons. The *ul* robot interacts with the *dl* robot through the *dl_pb1* pushbutton while the *pl* robot uses *dl_pb2*. Inside the *dl* robot there exists an engine (*dl_engine*) and an Abstract Controller Machine (*dl_acm*). No communication from the *dl* robot to the others is shown. It is intentionally left out to simplify the discussion.

StateMate Activity Chart

A StateMate model of the MachineChart of figure 6.6 based on the *Generalized Model for Behavioural Specification* of Chapter 4 is depicted in figure 6.7. The *ul* and *pl* robots are depicted as external activities while the *dl* robot as well as external

Figure 6.6: Link Layer Robot

Figure 6.7: Link Layer Activity Chart

interactions is described in detail. The subactivities of the *dl* activity represent the pushbuttons, engine and ACM. Concrete events, such as *cc_data_rq*, are buffered by the pushbutton activities and delivered to the engine activity as data (*dl_pb1_data*) when the engine is able to process them.

The concrete events associated with the *dl_pb1* pushbutton are:

- *cc_conn_rq* (*connect_request*)
- *cc_conn_rs* (*connect_response*)
- *cc_disc_rq* (*disconnect_request*)
- *cc_data_rq* (*data_request*)

The concrete events associated with the *dl_pb2* pushbutton are:

- *cc_open*
- *cc_close*
- *cc_data_0*
- *cc_data_1*
- *cc_ack_0*
- *cc_ack_1*

The engine interprets these concrete events and interacts with the ACM through the *abstract_event/abstract_action* interface to determine the next action.

Behaviour

The behaviour of the Abstract Controller Machine is represented by the StateChart of figure 6.3 described above.

The behaviour of the Engine is described by the StateChart of figure 6.8 with an exploded view of the Engine/Pushbutton interface in figure 6.9. It consists of a pushbutton interface (*dl_pb1,dl_pb2*), Abstract Controller Machine interface (*event*,

Figure 6.8: Link Layer Engine StateChart

Figure 6.9: Link Layer Engine/Pushbutton Interface

action) and core *engine* components. StateCharts for each pushbutton activity are included but not examined in detail having been explained in previous chapters.

The discussion below focuses on the behaviour of the *dl_engine* activity, its interaction with the surrounding activities (*dl_acm*, *dl_pb1*, *dl_pb2*), the translation of concrete events to *abstract events* and the processing of *abstract actions* returned from the ACM.

DL_ENGINE_CNTL StateChart

The *dl_engine_cntl* StateChart consists of the following components:

- *dl_pb1*

The interface which defines the service provided by the *link layer* to the *user* layer is defined by *dl_pb1*.

The default state is *closed*. The event *dl_pb1_enable* causes a transition to the *open* state. The first time this state is entered the event *rd(dl_pb1_data)* is generated and the *open.idle* state entered. The receipt of the *wr(dl_pb1_data)* event causes a transition to the *open.busy* state indicating that there is data for the *engine* component to process. If the *engine* component processes the data then a transition back to the *open.idle* state is made and the event *rd(dl_pb1_data)* generated to indicate a readiness for more input. In either the *open.idle* state or the *open.busy* state the reception of a *dl_pb1_disable* causes a transition to the *closed* state. The use of the *history connector* in the *open* state guarantees that the proper state, *open.idle* or *open.busy*, is reentered on the reception of a *dl_pb1_enable* event.

- *dl_pb2*

The interface which defines the service provided by the *physical layer* to the *link layer* is defined by *dl_pb2*.

The behaviour of the *dl_pb2* interface is identical to that of the *dl_pb1* interface.

- *event*

This component consists of two states; *idle*, the default state, and *delivered*. A transition to the *delivered* state is made when the *abstract_event* is generated

to the *dl_acm* activity. This also causes the event *rd(abstract_action)* to be generated to allow the *dl_acm* activity to return an *abstract_action*. The *idle* state is returned to when the *dl_acm* activity is ready to accept more input indicated by the event *rd(abstract_event)*.

- *action*

This component controls the reception of an *abstract_action* from the *dl_acm* activity. The default state is *idle* with a transition being made to the *available* state when an *abstract_action* is returned. The *idle* state is returned to when the *abstract_action* has been processed by the *engine* component indicated by the exit from the *engine_pre_acm_action* state.

- *engine*

The *engine* component cycles through a number of states classified as follows:

- *wait for concrete events*
- *interpret concrete events and generate abstract event*
- *wait for acm*
- *process acm action*

The default state is *wt_pushbuttons*. Upon entry to this state for the first time the action *dl_engine_init* is executed which enables the reception of concrete events via the pushbutton interface by generating the events *dl_pb1_enable* and *dl_pb2_enable*.

The availability of concrete events causes a transition to either the *pre_pb1_input* state or the *pre_pb2_input* state. At this point the SCP takes over, converting the concrete event to an abstract event then sending it to the *dl_acm* activity by generating the event *wr(abstract_event)*. The event *scp_done* indicates that the SCP is finished and causes a transition to the *wt_acm* state.

The reception of the *abstract_action* causes a transition to the *pre_acm_action* state at which point the SCP takes over once again to perform the actual

processing. This involves the generation of external events (not shown) and the opening or closing of pushbutton interfaces.

Upon SCP completion (*scp_done*) a transition to the *wt_pushbuttons* state is made to wait for new concrete events.

DL_PB1 StateChart

The *dl_pb1_cntl* StateChart is shown in figure 6.10.

Details of pushbutton behaviour are provided in the *Generalized Model for Behavioural Specification* in Chapter 4 and is summarized below.

Pushbuttons are implemented as queues using two buffers for input and output which are represented by the *put_q* and *get_q* components of the associated StateChart. Concrete events are detected by the *put_q* component; converted to integer values and then placed in the input buffer. A request to retrieve an item from the queue is detected by the *get_q* component. If an item is available in the output buffer it is retrieved directly. If the output buffer is empty and the input buffer is not then the buffers are swapped and the item retrieved. Should no input be available in either buffer then processing of the request is delayed until the reception of a concrete event.

DL_PB2_CNTL StateChart

The *dl_pb2_cntl* StateChart is shown in figure 6.11. The behaviour is similar to that of the *dl_pb1_cntl* StateChart.

6.5 Discussion

The StateMate interface implemented for the Link Layer between the ACM (Abstract Controller Machine) and the engine consists of the exchange of events and actions. Since no data is exchanged, numbered frames and acknowledgements must all have unique events associated with them. The MachineCharts interface, however, does allow for data to be exchanged between engine and ACM. By extending the ACM/Engine interface to include the equivalent of state data (state variables),

Figure 6.10: Link Layer Pushbutton 1 StateChart

Figure 6.11: Link Layer Pushbutton 2 StateChart

Figure 6.12: Multiple Link Layer Connections

state explosions may be avoided. Just as concrete events are mapped to abstract events, concrete data is mapped to abstract data. This extension permits the specification of a single ACM for the *Go-Back-N* link layer protocol for arbitrary value of N rather than require a different specification for each ACM. This approach is an advantage in layered protocols where the value of N representing the window size is negotiated between peer layer entities when the layer connection is established.

Multiple data link layer connections may be supported using the same ACM design specified above to manage individual data link layer connections and defining a new functional component to manage the connections as a group. Figure 6.12 illustrates one possible solution. The *Connection Manager* receives requests for connection establishment, both local and remote, and creates or assigns a *Link Manager* to manage the particular link. All subsequent requests (local and remote) are handled directly by the *Link Manager* until the connection is terminated. This is an example of the benefit of employing *Abstract Data* as an additional interface element between the Engine and the ACM since potentially each layer connection may support a different window size N .

Chapter 7

Conclusions

7.1 Accomplishments

This thesis provides a StateMate model of key MachineCharts components which allows StateMate to be used as a tool for design. The abstract behaviour (i.e. temporal aspects) of the system is specified using StateCharts and the StateMate tools used to analyse, verify and animate the behaviour of the design. Due to the complexity of the model it is more suitable for small design problems rather than large ones. The following benefits are perceived:

- Analysis and Verification of Abstract Behaviour

The structure of the model allows the StateMate toolset to be utilized in the analysis and verification of abstract behaviour. Tests for correctness, completeness, race conditions and deadlock, for example, may be carried out at the design phase of the software life cycle allowing errors to be detected and corrected early on with minimal impact on the remainder of the design. Once satisfied that the specification is correct the StateMate Prototyper may be used to automatically translate the specification into Ada code. This in turn may be compiled and linked with the target system and the abstract behaviour verified with the real system.

- Implementation Independent

The nature of the model provides a clear separation between abstract behaviour and structure and thus implies structural, or implementation independence. The MachineCharts notation is just one of many that could be used to specify the structure of the surrounding system into which the abstract behaviour specification may be integrated.

Additional contributions include:

- **Solution to Queuing in StateMate**

StateMate has its roots in a broadcast, transient event model which differs from that of other state machine models. This thesis provides a means to specify a queue in StateMate so that events are not lost. This was a crucial step in the alignment procedure and provided the “glue” between the broadcast, transient event model of StateMate and the queued, directed event model of MachineCharts.

- **Design Methodology**

The case studies demonstrate an effective methodology for applying the model to different design problems.

- **Style Guide for StateMate Specifications**

The development and application of the model provides a useful tutorial of how to use StateMate when tackling new design problems.

7.2 Future Work

The following is a list of topics for future work:

- **Develop a better model of a MachineCharts gate button.**

The StateMate queue activity developed in this thesis has characteristics similar to that of a MachineCharts gate button but because the buffering of events is only one way, it more closely resembles a mailbox used in message passing operating systems than the MachineCharts gate button which is based on the rendezvous principle of Ada.

The development of a StateMate button activity based on the principles of the StateMate queue activity would improve the degree of alignment between the MachineCharts and StateMate notations.

- Investigate the use of the upcoming StateMate Graphical Prototyper to provide a MachineCharts visual representation of the animation of a MachineCharts design which is expressed in StateMate terms based on the StateMate Model of MachineCharts Components.

This would involve specifying MachineCharts icons for each of the components in the model with the Graphical Prototyper and having the animation of these icons driven by the output of a simulation of the design using the StateMate Analyser.

- Investigate the use of the StateMate animation software to drive the animation of a MachineCharts design within the TimeBench environment.

In other words, investigate the integration of the StateMate animation software with the TimeBench software.

References

- [BHK⁺86] R.J.A. Buhr, C.J. Hayes, G.M. Karam, E. Neron, and R. Casselman. CAEDE 1.5 User's Guide. Technical report, Dept. of Systems and Computer Engineering, Carleton University, May 1986.
- [BK85] R.J.A. Buhr and G.M. Karam. An Informal Overview of CAEDE. Technical report, Dept. of Systems and Computer Engineering, Carleton University, March/April 1985.
- [BKC90] R.J.A. Buhr, G.M. Karam, and R. Casselman. Support for Specifying Temporal Behaviour in Ada Designs. Technical report, Dept. of Systems and Computer Engineering, Carleton University, January 1990.
- [BKHW89] R.J.A. Buhr, G.M. Karam, C.J. Hayes, and C.M. Woodside. Software CAD: A Revolutionary Approach. Technical report, Dept. of Systems and Computer Engineering, Carleton University, March 1989.
- [BKW85a] R.J.A. Buhr, G.M. Karam, and C.M. Woodside. An Overview and Example Application of CAEDE: A New Design Environment for Ada. Technical report, Dept. of Systems and Computer Engineering, Carleton University, May 1985.
- [BKW⁺85b] R.J.A. Buhr, G.M. Karam, C.M. Woodside, K. Van Der Loo, and D.G. Lewis. Experiments With Prolog Design Descriptions and Tools in CAEDE: An Iconic Design Environment for Multitasking Embedded Systems. Technical report, Dept. of Systems and Computer Engineering, Carleton University, May 1985.
- [Buh84] Buhr. *System Design With Ada*. Prentice Hall, Englewood Cliffs, New Jersey, first edition, 1984.
- [Buh88] R.J.A. Buhr. Draft - MachineCharts: An Iconic Notation for Software Design. Technical report, Dept. of Systems and Computer Engineering, Carleton University, January 1988.
- [Buh90a] Buhr. *Practical Visual Techniques in System Design with Applications to Ada*. Prentice Hall, Englewood Cliffs, New Jersey, first edition, 1990.

- [Buh90b] RJA Buhr. Practical Visual Techniques For Joint Refinement of Structure and Temporal Behaviour in Reactive System Design. Technical report, Dept. of Systems and Computer Engineering, Carleton University, January 1990.
- [Har88] David Harel. StateMate: A Working Environment For Complex Reactive Systems. *Proceedings of the 10th International Conference on Software Engineering*, April 1988.
- [Hou79] Trevor Housley. *Data Communications and Teleprocessing Systems*. Prentice Hall, Englewood Cliffs, New Jersey, first edition, 1979.
- [iL87] i Logix. *The Languages of StateMate*. i-Logix, Boston, first edition, 1987.
- [iL88] i Logix. *StateMate Kernel Sun Ver. 2.0*. i-Logix, Boston, first edition, 1988.
- [KBWC89] G.M. Karam, R.J.A. Buhr, C.M. Woodside, and R. Casselman. TimeBench: A Next Generation CASE Tool. Technical report, Dept of Systems and Computer Engineering, Carleton University, July 1989.
- [Sta85] William Stallings. *Data and Computer Communications*. Macmillan Publishing Company, New York, New York, first edition, 1985.

Appendix A

Glossary of Terms

- **Abstract Action**

An abstract action is returned from an Abstract Controller Machine in response to an abstract event.

- **Abstract Behaviour**

Abstract behaviour refers to the behaviour of a system from a requirements standpoint without consideration of design or implementation.

- **Abstract Controller Machine**

An Abstract Controller Machine is a MachineCharts component which encapsulates abstract behaviour. When given an abstract event, it responds with an abstract action which is used by an engine to determine what action to perform next.

- **Abstract Event**

An abstract event is the input to an Abstract Controller Machine which responds with an abstract action.

- **ActivityCharts**

ActivityCharts is a StateMate graphical notation which is used to specify the functional components of a system and the flow of information between them.

An ActivityChart refers to a single diagram constructed with the Activity Charts graphical editor.

- **Animation**

Animation refers to the visual representation of a requirements or design simulation.

With respect to StateCharts, the animation of behaviour is presented through the successive highlighting of states and transitions.

- **Box**

A box is a MachineCharts component which represents packaging.

- **Broadcast, Transient Events**

This is the event model upon which StateMate is based.

Events are broadcast to all activities in the system and are transient in nature as opposed to queued. If events are not processed when they occur they are lost.

- **Button - Work and Gate**

Work and gate buttons represent two types of interactions in the MachineCharts notation.

Work buttons imply that the activities associated with the button are carried out immediately.

Gate buttons, on the other hand, imply that synchronization is required between two MachineCharts components before the activities are performed.

Work and gate buttons are attached to boxes and robots respectively to define the interface seen by other components in the system.

- **Concrete Action**

A concrete action is associated with the interactions between the design components of a system. An example of a concrete action is the action of visiting a MachineCharts gate button.

- **Concrete Behaviour**

Concrete behaviour refers to the behaviour of a system from a design standpoint.

- **Concrete Event**

A concrete event is associated with the interactions between design components of a system. For example, a visit to a MachineCharts gate button represents a concrete event.

- **Engine**

Engines define the activity of the components (e.g. robots, buttons) in which they reside.

- **Event Model**

An event model is another term for the communication semantics between concurrent entities in a reactive system

The two event models discussed in this thesis are queued, directed events and broadcast transient events.

- **Finger**

A finger is a directed line between two MachineCharts components and represents the potential for interaction.

- **Functional**

The term functional refers to the activities which take place in a system (i.e. what the system does).

- **MachineCharts**

MachineCharts is a graphical design notation developed by Buhr to describe the functional and structural aspects of a system.

- **Queued, Directed Events**

This is the event model upon which MachineCharts is based.

Events are directed between MachineCharts components and are queued until they are processed.

- **Reactive System**

A reactive system is an event driven system whose output is determined by input consisting of a set of events, the time at which these events occur and the time to process these events.

- **Robot**

A robot is a MachineCharts component which represents a unit of concurrency in a system.

- **Simulation Control Program**

A Simulation Control Program is a program written by a StateMate user to automate the simulation of a StateMate specification.

- **StateCharts**

StateCharts is a StateMate graphical notation which is used to specify the temporal aspects of a system including states, event, conditions and actions.

A StateChart refers to a single diagram constructed with the StateCharts graphical editor.

- **StateMate**

StateMate is a commercially available CASE tool which allows the functional and temporal aspects of a system to be described, analysed, verified and simulated.

- **StateMate Analyser**

The StateMate Analyser consists of a collection of tools which provide a means to analyse the specification of a system and verify its correct operation.

- **Structural**

The term structural refers to the packaging of functional and temporal aspects of system. The structure of a system denotes the pieces which make up the system and how they fit together.

- **Temporal**

The term temporal refers to the time when events and actions occur in a system.

- **Visit**

A visit refers to an attempt by one MachineCharts component to interact with another.

Appendix B

Implementation examples of Resource Manager

This appendix contains the example code for the readers and writers case study.

There are four files as follows:

- Abstract Controller Machine
- Resource Manager Task for Example 1
- Resource Manager Task for Example 2
- Resource Manager Task for Example 3

```

1  parallel unit acm.
2  INTERFACE
3  {
4    this module provides the Abstract Controller Machine for
5    the resource manager of the readers and writers case
6    study.
7  }
8  TYPE
9
10     lgc1_evt = (l_get_rd_access,
11                l_get_wrt_access,
12                l_rls_access);
13
14     lgc1_actn = (l_nil,
15                 l_rdr_actv,
16                 l_rdrs_actv,
17                 l_rdr_pnd_rls,
18                 l_wtr_actv,
19                 l_wtr_pnd_rls,
20                 l_rdr_rls_actv,
21                 l_rdr_rls_rsrc_avl,
22                 l_wtr_rls_rsrc_avl);
23
24     FUNCTION prc_lgc1_evt(evt : lgc1_evt) : lgc1_actn;
25     IMPLEMENTATION
26
27     FUNCTION prc_lgc1_evt(evt : lgc1_evt) : lgc1_actn;
28
29     BEGIN
30       {
31         Refer to state chart for behaviour.
32       }
33       prc_lgc1_evt := l_nil;
34     END;
35
36     size : 46
37
38
39     BEGIN
40       {
41         Refer to state chart for behaviour.
42       }
43       prc_lgc1_evt := l_nil;
44     END;
45
46     size : 28
47
48     BEGIN
49       {
50         Refer to state chart for behaviour.
51       }
52       prc_lgc1_evt := l_nil;
53     END;
54
55     size : 28
56
57     BEGIN
58       {
59         Refer to state chart for behaviour.
60       }
61       prc_lgc1_evt := l_nil;
62     END;
63
64     size : 28
65
66     BEGIN
67       {
68         Refer to state chart for behaviour.
69       }
70       prc_lgc1_evt := l_nil;
71     END;
72
73     size : 28
74
75     BEGIN
76       {
77         Refer to state chart for behaviour.
78       }
79       prc_lgc1_evt := l_nil;
80     END;
81
82     size : 28
83
84     BEGIN
85       {
86         Refer to state chart for behaviour.
87       }
88       prc_lgc1_evt := l_nil;
89     END;
90
91     size : 28
92
93     BEGIN
94       {
95         Refer to state chart for behaviour.
96       }
97       prc_lgc1_evt := l_nil;
98     END;
99
100    size : 28
101
102    BEGIN
103      {
104        Refer to state chart for behaviour.
105      }
106      prc_lgc1_evt := l_nil;
107    END;
108
109    size : 28
110
111    BEGIN
112      {
113        Refer to state chart for behaviour.
114      }
115      prc_lgc1_evt := l_nil;
116    END;
117
118    size : 28
119
120    BEGIN
121      {
122        Refer to state chart for behaviour.
123      }
124      prc_lgc1_evt := l_nil;
125    END;
126
127    size : 28
128
129    BEGIN
130      {
131        Refer to state chart for behaviour.
132      }
133      prc_lgc1_evt := l_nil;
134    END;
135
136    size : 28
137
138    BEGIN
139      {
140        Refer to state chart for behaviour.
141      }
142      prc_lgc1_evt := l_nil;
143    END;
144
145    size : 28
146
147    BEGIN
148      {
149        Refer to state chart for behaviour.
150      }
151      prc_lgc1_evt := l_nil;
152    END;
153
154    size : 28
155
156    BEGIN
157      {
158        Refer to state chart for behaviour.
159      }
160      prc_lgc1_evt := l_nil;
161    END;
162
163    size : 28
164
165    BEGIN
166      {
167        Refer to state chart for behaviour.
168      }
169      prc_lgc1_evt := l_nil;
170    END;
171
172    size : 28
173
174    BEGIN
175      {
176        Refer to state chart for behaviour.
177      }
178      prc_lgc1_evt := l_nil;
179    END;
180
181    size : 28
182
183    BEGIN
184      {
185        Refer to state chart for behaviour.
186      }
187      prc_lgc1_evt := l_nil;
188    END;
189
190    size : 28
191
192    BEGIN
193      {
194        Refer to state chart for behaviour.
195      }
196      prc_lgc1_evt := l_nil;
197    END;
198
199    size : 28
200
201    BEGIN
202      {
203        Refer to state chart for behaviour.
204      }
205      prc_lgc1_evt := l_nil;
206    END;
207
208    size : 28
209
210    BEGIN
211      {
212        Refer to state chart for behaviour.
213      }
214      prc_lgc1_evt := l_nil;
215    END;
216
217    size : 28
218
219    BEGIN
220      {
221        Refer to state chart for behaviour.
222      }
223      prc_lgc1_evt := l_nil;
224    END;
225
226    size : 28
227
228    BEGIN
229      {
230        Refer to state chart for behaviour.
231      }
232      prc_lgc1_evt := l_nil;
233    END;
234
235    size : 28
236
237    BEGIN
238      {
239        Refer to state chart for behaviour.
240      }
241      prc_lgc1_evt := l_nil;
242    END;
243
244    size : 28
245
246    BEGIN
247      {
248        Refer to state chart for behaviour.
249      }
250      prc_lgc1_evt := l_nil;
251    END;
252
253    size : 28
254
255    BEGIN
256      {
257        Refer to state chart for behaviour.
258      }
259      prc_lgc1_evt := l_nil;
260    END;
261
262    size : 28
263
264    BEGIN
265      {
266        Refer to state chart for behaviour.
267      }
268      prc_lgc1_evt := l_nil;
269    END;
270
271    size : 28
272
273    BEGIN
274      {
275        Refer to state chart for behaviour.
276      }
277      prc_lgc1_evt := l_nil;
278    END;
279
280    size : 28
281
282    BEGIN
283      {
284        Refer to state chart for behaviour.
285      }
286      prc_lgc1_evt := l_nil;
287    END;
288
289    size : 28
290
291    BEGIN
292      {
293        Refer to state chart for behaviour.
294      }
295      prc_lgc1_evt := l_nil;
296    END;
297
298    size : 28
299
300    BEGIN
301      {
302        Refer to state chart for behaviour.
303      }
304      prc_lgc1_evt := l_nil;
305    END;
306
307    size : 28
308
309    BEGIN
310      {
311        Refer to state chart for behaviour.
312      }
313      prc_lgc1_evt := l_nil;
314    END;
315
316    size : 28
317
318    BEGIN
319      {
320        Refer to state chart for behaviour.
321      }
322      prc_lgc1_evt := l_nil;
323    END;
324
325    size : 28
326
327    BEGIN
328      {
329        Refer to state chart for behaviour.
330      }
331      prc_lgc1_evt := l_nil;
332    END;
333
334    size : 28
335
336    BEGIN
337      {
338        Refer to state chart for behaviour.
339      }
340      prc_lgc1_evt := l_nil;
341    END;
342
343    size : 28
344
345    BEGIN
346      {
347        Refer to state chart for behaviour.
348      }
349      prc_lgc1_evt := l_nil;
350    END;
351
352    size : 28
353
354    BEGIN
355      {
356        Refer to state chart for behaviour.
357      }
358      prc_lgc1_evt := l_nil;
359    END;
360
361    size : 28
362
363    BEGIN
364      {
365        Refer to state chart for behaviour.
366      }
367      prc_lgc1_evt := l_nil;
368    END;
369
370    size : 28
371
372    BEGIN
373      {
374        Refer to state chart for behaviour.
375      }
376      prc_lgc1_evt := l_nil;
377    END;
378
379    size : 28
380
381    BEGIN
382      {
383        Refer to state chart for behaviour.
384      }
385      prc_lgc1_evt := l_nil;
386    END;
387
388    size : 28
389
390    BEGIN
391      {
392        Refer to state chart for behaviour.
393      }
394      prc_lgc1_evt := l_nil;
395    END;
396
397    size : 28
398
399    BEGIN
400      {
401        Refer to state chart for behaviour.
402      }
403      prc_lgc1_evt := l_nil;
404    END;
405
406    size : 28
407
408    BEGIN
409      {
410        Refer to state chart for behaviour.
411      }
412      prc_lgc1_evt := l_nil;
413    END;
414
415    size : 28
416
417    BEGIN
418      {
419        Refer to state chart for behaviour.
420      }
421      prc_lgc1_evt := l_nil;
422    END;
423
424    size : 28
425
426    BEGIN
427      {
428        Refer to state chart for behaviour.
429      }
430      prc_lgc1_evt := l_nil;
431    END;
432
433    size : 28
434
435    BEGIN
436      {
437        Refer to state chart for behaviour.
438      }
439      prc_lgc1_evt := l_nil;
440    END;
441
442    size : 28
443
444    BEGIN
445      {
446        Refer to state chart for behaviour.
447      }
448      prc_lgc1_evt := l_nil;
449    END;
450
451    size : 28
452
453    BEGIN
454      {
455        Refer to state chart for behaviour.
456      }
457      prc_lgc1_evt := l_nil;
458    END;
459
460    size : 28
461
462    BEGIN
463      {
464        Refer to state chart for behaviour.
465      }
466      prc_lgc1_evt := l_nil;
467    END;
468
469    size : 28
470
471    BEGIN
472      {
473        Refer to state chart for behaviour.
474      }
475      prc_lgc1_evt := l_nil;
476    END;
477
478    size : 28
479
480    BEGIN
481      {
482        Refer to state chart for behaviour.
483      }
484      prc_lgc1_evt := l_nil;
485    END;
486
487    size : 28
488
489    BEGIN
490      {
491        Refer to state chart for behaviour.
492      }
493      prc_lgc1_evt := l_nil;
494    END;
495
496    size : 28
497
498    BEGIN
499      {
500        Refer to state chart for behaviour.
501      }
502      prc_lgc1_evt := l_nil;
503    END;
504
505    size : 28
506
507    BEGIN
508      {
509        Refer to state chart for behaviour.
510      }
511      prc_lgc1_evt := l_nil;
512    END;
513
514    size : 28
515
516    BEGIN
517      {
518        Refer to state chart for behaviour.
519      }
520      prc_lgc1_evt := l_nil;
521    END;
522
523    size : 28
524
525    BEGIN
526      {
527        Refer to state chart for behaviour.
528      }
529      prc_lgc1_evt := l_nil;
530    END;
531
532    size : 28
533
534    BEGIN
535      {
536        Refer to state chart for behaviour.
537      }
538      prc_lgc1_evt := l_nil;
539    END;
540
541    size : 28
542
543    BEGIN
544      {
545        Refer to state chart for behaviour.
546      }
547      prc_lgc1_evt := l_nil;
548    END;
549
550    size : 28
551
552    BEGIN
553      {
554        Refer to state chart for behaviour.
555      }
556      prc_lgc1_evt := l_nil;
557    END;
558
559    size : 28
560
561    BEGIN
562      {
563        Refer to state chart for behaviour.
564      }
565      prc_lgc1_evt := l_nil;
566    END;
567
568    size : 28
569
570    BEGIN
571      {
572        Refer to state chart for behaviour.
573      }
574      prc_lgc1_evt := l_nil;
575    END;
576
577    size : 28
578
579    BEGIN
580      {
581        Refer to state chart for behaviour.
582      }
583      prc_lgc1_evt := l_nil;
584    END;
585
586    size : 28
587
588    BEGIN
589      {
590        Refer to state chart for behaviour.
591      }
592      prc_lgc1_evt := l_nil;
593    END;
594
595    size : 28
596
597    BEGIN
598      {
599        Refer to state chart for behaviour.
600      }
601      prc_lgc1_evt := l_nil;
602    END;
603
604    size : 28
605
606    BEGIN
607      {
608        Refer to state chart for behaviour.
609      }
610      prc_lgc1_evt := l_nil;
611    END;
612
613    size : 28
614
615    BEGIN
616      {
617        Refer to state chart for behaviour.
618      }
619      prc_lgc1_evt := l_nil;
620    END;
621
622    size : 28
623
624    BEGIN
625      {
626        Refer to state chart for behaviour.
627      }
628      prc_lgc1_evt := l_nil;
629    END;
630
631    size : 28
632
633    BEGIN
634      {
635        Refer to state chart for behaviour.
636      }
637      prc_lgc1_evt := l_nil;
638    END;
639
640    size : 28
641
642    BEGIN
643      {
644        Refer to state chart for behaviour.
645      }
646      prc_lgc1_evt := l_nil;
647    END;
648
649    size : 28
650
651    BEGIN
652      {
653        Refer to state chart for behaviour.
654      }
655      prc_lgc1_evt := l_nil;
656    END;
657
658    size : 28
659
660    BEGIN
661      {
662        Refer to state chart for behaviour.
663      }
664      prc_lgc1_evt := l_nil;
665    END;
666
667    size : 28
668
669    BEGIN
670      {
671        Refer to state chart for behaviour.
672      }
673      prc_lgc1_evt := l_nil;
674    END;
675
676    size : 28
677
678    BEGIN
679      {
680        Refer to state chart for behaviour.
681      }
682      prc_lgc1_evt := l_nil;
683    END;
684
685    size : 28
686
687    BEGIN
688      {
689        Refer to state chart for behaviour.
690      }
691      prc_lgc1_evt := l_nil;
692    END;
693
694    size : 28
695
696    BEGIN
697      {
698        Refer to state chart for behaviour.
699      }
700      prc_lgc1_evt := l_nil;
701    END;
702
703    size : 28
704
705    BEGIN
706      {
707        Refer to state chart for behaviour.
708      }
709      prc_lgc1_evt := l_nil;
710    END;
711
712    size : 28
713
714    BEGIN
715      {
716        Refer to state chart for behaviour.
717      }
718      prc_lgc1_evt := l_nil;
719    END;
720
721    size : 28
722
723    BEGIN
724      {
725        Refer to state chart for behaviour.
726      }
727      prc_lgc1_evt := l_nil;
728    END;
729
730    size : 28
731
732    BEGIN
733      {
734        Refer to state chart for behaviour.
735      }
736      prc_lgc1_evt := l_nil;
737    END;
738
739    size : 28
740
741    BEGIN
742      {
743        Refer to state chart for behaviour.
744      }
745      prc_lgc1_evt := l_nil;
746    END;
747
748    size : 28
749
750    BEGIN
751      {
752        Refer to state chart for behaviour.
753      }
754      prc_lgc1_evt := l_nil;
755    END;
756
757    size : 28
758
759    BEGIN
760      {
761        Refer to state chart for behaviour.
762      }
763      prc_lgc1_evt := l_nil;
764    END;
765
766    size : 28
767
768    BEGIN
769      {
770        Refer to state chart for behaviour.
771      }
772      prc_lgc1_evt := l_nil;
773    END;
774
775    size : 28
776
777    BEGIN
778      {
779        Refer to state chart for behaviour.
780      }
781      prc_lgc1_evt := l_nil;
782    END;
783
784    size : 28
785
786    BEGIN
787      {
788        Refer to state chart for behaviour.
789      }
790      prc_lgc1_evt := l_nil;
791    END;
792
793    size : 28
794
795    BEGIN
796      {
797        Refer to state chart for behaviour.
798      }
799      prc_lgc1_evt := l_nil;
800    END;
801
802    size : 28
803
804    BEGIN
805      {
806        Refer to state chart for behaviour.
807      }
808      prc_lgc1_evt := l_nil;
809    END;
810
811    size : 28
812
813    BEGIN
814      {
815        Refer to state chart for behaviour.
816      }
817      prc_lgc1_evt := l_nil;
818    END;
819
820    size : 28
821
822    BEGIN
823      {
824        Refer to state chart for behaviour.
825      }
826      prc_lgc1_evt := l_nil;
827    END;
828
829    size : 28
830
831    BEGIN
832      {
833        Refer to state chart for behaviour.
834      }
835      prc_lgc1_evt := l_nil;
836    END;
837
838    size : 28
839
840    BEGIN
841      {
842        Refer to state chart for behaviour.
843      }
844      prc_lgc1_evt := l_nil;
845    END;
846
847    size : 28
848
849    BEGIN
850      {
851        Refer to state chart for behaviour.
852      }
853      prc_lgc1_evt := l_nil;
854    END;
855
856    size : 28
857
858    BEGIN
859      {
860        Refer to state chart for behaviour.
861      }
862      prc_lgc1_evt := l_nil;
863    END;
864
865    size : 28
866
867    BEGIN
868      {
869        Refer to state chart for behaviour.
870      }
871      prc_lgc1_evt := l_nil;
872    END;
873
874    size : 28
875
876    BEGIN
877      {
878        Refer to state chart for behaviour.
879      }
880      prc_lgc1_evt := l_nil;
881    END;
882
883    size : 28
884
885    BEGIN
886      {
887        Refer to state chart for behaviour.
888      }
889      prc_lgc1_evt := l_nil;
890    END;
891
892    size : 28
893
894    BEGIN
895      {
896        Refer to state chart for behaviour.
897      }
898      prc_lgc1_evt := l_nil;
899    END;
900
901    size : 28
902
903    BEGIN
904      {
905        Refer to state chart for behaviour.
906      }
907      prc_lgc1_evt := l_nil;
908    END;
909
910    size : 28
911
912    BEGIN
913      {
914        Refer to state chart for behaviour.
915      }
916      prc_lgc1_evt := l_nil;
917    END;
918
919    size : 28
920
921    BEGIN
922      {
923        Refer to state chart for behaviour.
924      }
925      prc_lgc1_evt := l_nil;
926    END;
927
928    size : 28
929
930    BEGIN
931      {
932        Refer to state chart for behaviour.
933      }
934      prc_lgc1_evt := l_nil;
935    END;
936
937    size : 28
938
939    BEGIN
940      {
941        Refer to state chart for behaviour.
942      }
943      prc_lgc1_evt := l_nil;
944    END;
945
946    size : 28
947
948    BEGIN
949      {
950        Refer to state chart for behaviour.
951      }
952      prc_lgc1_evt := l_nil;
953    END;
954
955    size : 28
956
957    BEGIN
958      {
959        Refer to state chart for behaviour.
960      }
961      prc_lgc1_evt := l_nil;
962    END;
963
964    size : 28
965
966    BEGIN
967      {
968        Refer to state chart for behaviour.
969      }
970      prc_lgc1_evt := l_nil;
971    END;
972
973    size : 28
974
975    BEGIN
976      {
977        Refer to state chart for behaviour.
978      }
979      prc_lgc1_evt := l_nil;
980    END;
981
982    size : 28
983
984    BEGIN
985      {
986        Refer to state chart for behaviour.
987      }
988      prc_lgc1_evt := l_nil;
989    END;
990
991    size : 28
992
993    BEGIN
994      {
995        Refer to state chart for behaviour.
996      }
997      prc_lgc1_evt := l_nil;
998    END;
999
1000   size : 28
1001
1002   BEGIN
1003     {
1004       Refer to state chart for behaviour.
1005     }
1006     prc_lgc1_evt := l_nil;
1007   END;
1008
1009   size : 28
1010
1011   BEGIN
1012     {
1013       Refer to state chart for behaviour.
1014     }
1015     prc_lgc1_evt := l_nil;
1016   END;
1017
1018   size : 28
1019
1020   BEGIN
1021     {
1022       Refer to state chart for behaviour.
1023     }
1024     prc_lgc1_evt := l_nil;
1025   END;
1026
1027   size : 28
1028
1029   BEGIN
1030     {
1031       Refer to state chart for behaviour.
1032     }
1033     prc_lgc1_evt := l_nil;
1034   END;
1035
1036   size : 28
1037
1038   BEGIN
1039     {
1040       Refer to state chart for behaviour.
1041     }
1042     prc_lgc1_evt := l_nil;
1043   END;
1044
1045   size : 28
1046
1047   BEGIN
1048     {
1049       Refer to state chart for behaviour.
1050     }
1051     prc_lgc1_evt := l_nil;
1052   END;
1053
1054   size : 28
1055
1056   BEGIN
1057     {
1058       Refer to state chart for behaviour.
1059     }
1060     prc_lgc1_evt := l_nil;
1061   END;
1062
1063   size : 28
1064
1065   BEGIN
1066     {
1067       Refer to state chart for behaviour.
1068     }
1069     prc_lgc1_evt := l_nil;
1070   END;
1071
1072   size : 28
1073
1074   BEGIN
1075     {
1076       Refer to state chart for behaviour.
1077     }
1078     prc_lgc1_evt := l_nil;
1079   END;
1080
1081   size : 28
1082
1083   BEGIN
1084     {
1085       Refer to state chart for behaviour.
1086     }
1087     prc_lgc1_evt := l_nil;
1088   END;
1089
1090   size : 28
1091
1092   BEGIN
1093     {
1094       Refer to state chart for behaviour.
1095     }
1096     prc_lgc1_evt := l_nil;
1097   END;
1098
1099   size : 28
1100
1101   BEGIN
1102     {
1103       Refer to state chart for behaviour.
1104     }
1105     prc_lgc1_evt := l_nil;
1106   END;
1107
1108   size : 28
1109
1110   BEGIN
1111     {
1112       Refer to state chart for behaviour.
1113     }
1114     prc_lgc1_evt := l_nil;
1115   END;
1116
1117   size : 28
1118
1119   BEGIN
1120     {
1121       Refer to state chart for behaviour.
1122     }
1123     prc_lgc1_evt := l_nil;
1124   END;
1125
1126   size : 28
1127
1128   BEGIN
1129     {
1130       Refer to state chart for behaviour.
1131     }
1132     prc_lgc1_evt := l_nil;
1133   END;
1134
1135   size : 28
1136
1137   BEGIN
1138     {
1139       Refer to state chart for behaviour.
1140     }
1141     prc_lgc1_evt := l_nil;
1142   END;
1143
1144   size : 28
1145
1146   BEGIN
1147     {
1148       Refer to state chart for behaviour.
1149     }
1150     prc_lgc1_evt := l_nil;
1151   END;
1152
1153   size : 28
1154
1155   BEGIN
1156     {
1157       Refer to state chart for behaviour.
1158     }
1159     prc_lgc1_evt := l_nil;
1160   END;
1161
1162   size : 28
1163
1164   BEGIN
1165     {
1166       Refer to state chart for behaviour.
1167     }
1168     prc_lgc1_evt := l_nil;
1169   END;
1170
1171   size : 28
1172
1173   BEGIN
1174     {
1175       Refer to state chart for behaviour.
1176     }
1177     prc_lgc1_evt := l_nil;
1178   END;
1179
1180   size : 28
1181
1182   BEGIN
1183     {
1184       Refer to state chart for behaviour.
1185     }
1186     prc_lgc1_evt := l_nil;
1187   END;
1188
1189   size : 28
1190
1191   BEGIN
1192     {
1193       Refer to state chart for behaviour.
1194     }
1195     prc_lgc1_evt := l_nil;
1196   END;
1197
1198   size : 28
1199
1200   BEGIN
1201     {
1202       Refer to state chart for behaviour.
1203     }
1204     prc_lgc1_evt := l_nil;
1205   END;
1206
1207   size : 28
1208
1209   BEGIN
1210     {
1211       Refer to state chart for behaviour.
1212     }
1213     prc_lgc1_evt := l_nil;
1214   END;
1215
1216   size : 28
1217
1218   BEGIN
1219     {
1220       Refer to state chart for behaviour.
1221     }
1222     prc_lgc1_evt := l_nil;
1223   END;
1224
1225   size : 28
1226
1227   BEGIN
1228     {
1229       Refer to state chart for behaviour.
1230     }
1231     prc_lgc1_evt := l_nil;
1232   END;
1233
1234   size : 28
1235
1236   BEGIN
1237     {
1238       Refer to state chart for behaviour.
1239     }
1240     prc_lgc1_evt := l_nil;
1241   END;
1242
1243   size : 28
1244
1245   BEGIN
1246     {
1247       Refer to state chart for behaviour.
1248     }
1249     prc_lgc1_evt := l_nil;
1250   END;
1251
1252   size : 28
1253
1254   BEGIN
1255     {
1256       Refer to state chart for behaviour.
1257     }
1258     prc_lgc1_evt := l_nil;
1259   END;
1260
1261   size : 28
1262
1263   BEGIN
1264     {
1265       Refer to state chart for behaviour.
1266     }
1267     prc_lgc1_evt := l_nil;
1268   END;
1269
1270   size : 28
1271
1272   BEGIN
1273     {
1274       Refer to state chart for behaviour.
1275     }
1276     prc_lgc1_evt := l_nil;
1277   END;
1278
1279   size : 28
1280
1281   BEGIN
1282     {
1283       Refer to state chart for behaviour.
1284     }
1285     prc_lgc1_evt := l_nil;
1286   END;
1287
1288   size : 28
1289
1290   BEGIN
1291     {
1292       Refer to state chart for behaviour.
1293     }
1294     prc_lgc1_evt := l_nil;
1295   END;
1296
1297   size : 28
1298
1299   BEGIN
1300     {
1301       Refer to state chart for behaviour.
1302     }
1303     prc_lgc1_evt := l_nil;
1304   END;
1305
1306   size : 28
1307
1308   BEGIN
1309     {
1310       Refer to state chart for behaviour.
1311     }
1312     prc_lgc1_evt := l_nil;
1313   END;
1314
1315   size : 28
1316
1317   BEGIN
1318     {
1319       Refer to state chart for behaviour.
1320     }
1321     prc_lgc1_evt := l_nil;
1322   END;
1323
1324   size : 28
1325
1326   BEGIN
1327     {
1328       Refer to state chart for behaviour.
1329     }
1330     prc_lgc1_evt := l_nil;
1331   END;
1332
1333   size : 28
1334
1335   BEGIN
1336     {
1337       Refer to state chart for behaviour.
1338     }
1339     prc_lgc1_evt := l_nil;
1340   END;
1341
1342   size : 28
1343
1344   BEGIN
1345     {
1346       Refer to state chart for behaviour.
1347     }
1348     prc_lgc1_evt := l_nil;
1349   END;
1350
1351   size : 28
1352
1353   BEGIN
1354     {
1355       Refer to state chart for behaviour.
1356     }
1357     prc_lgc1_evt := l_nil;
1358   END;
1359
1360   size : 28
1361
1362   BEGIN
1363     {
1364       Refer to state chart for behaviour.
1365     }
1366     prc_lgc1_evt := l_nil;
1367   END;
1368
1369   size : 28
1370
1371   BEGIN
1372     {
1373       Refer to state chart for behaviour.
1374     }
1375     prc_lgc1_evt := l_nil;
1376   END;
1377
1378   size : 28
1379
1380   BEGIN
1381     {
1382       Refer to state chart for behaviour.
1383     }
1384     prc_lgc1_evt := l_nil;
1385   END;
1386
1387   size : 28
1388
1389   BEGIN
1390     {
1391       Refer to state chart for behaviour.
1392     }
1393     prc_lgc1_evt := l_nil;
1394   END;
1395
1396   size : 28
1397
1398   BEGIN
1399     {
1400       Refer to state chart for behaviour.
1401     }
1402     prc_lgc1_evt := l_nil;
1403   END;
1404
1405   size : 28
1406
1407   BEGIN
1408     {
1409       Refer to state chart for behaviour.
1410     }
1411     prc_lgc1_evt := l_nil;
1412   END;
1413
1414   size : 28
1415
1416   BEGIN
1417     {
1418       Refer to state chart for behaviour.
1419     }
1420     prc_lgc1_evt := l_nil;
1421   END;
1422
1423   size : 28
1424
1425   BEGIN
1426     {
1427       Refer to state chart for behaviour.
1428     }
1429     prc_lgc1_evt := l_nil;
1430   END;
1431
1432   size : 28
1433
1434   BEGIN
1435     {
1436       Refer to state chart for behaviour.
1437     }
1438     prc_lgc1_evt := l_nil;
1439   END;
1440
1441   size : 28
1442
1443   BEGIN
1444     {
1445       Refer to state chart for behaviour.
1446     }
1447     prc_lgc1_evt := l_nil;
1448   END;
1449
1450   size : 28
1451
1452   BEGIN
1453     {
1454       Refer to state chart for behaviour.
1455     }
1456     prc_lgc1_evt := l_nil;
1457   END;
1458
1459   size : 28
1460
1461   BEGIN
1462     {
1463       Refer to state chart for behaviour.
1464     }
1465     prc_lgc1_evt := l_nil;
1466   END;
1467
1468   size : 28
1469
1470   BEGIN
1471     {
1472       Refer to state chart for behaviour.
1473     }
1474     prc_lgc1_evt := l_nil;
1475   END;
1476
1477   size : 28
1478
1479   BEGIN
1480     {
1481       Refer to state chart for behaviour.
1482     }
1483     prc_lgc1_evt := l_nil;
1484   END;
1485
1486   size : 28
1487
1488   BEGIN
1489     {
1490       Refer to state chart for behaviour.
1491     }
1492     prc_lgc1_evt := l_nil;
1493   END;
1494
1495   size : 28
1496
1497   BEGIN
1498     {
1499       Refer to state chart for behaviour.
1500     }
1501     prc_lgc1_evt := l_nil;
1502   END;
1503
1504   size : 28
1505
1506   BEGIN
1507     {
1508       Refer to state chart for behaviour.
1509     }
1510     prc_lgc1_evt := l_nil;
1511   END;
1512
1513   size : 28
1514
1515   BEGIN
1516     {
1517      
```



```

46      r_ri's_access
47      phy_to_lgcl_evt = l_ri's_access;
48      END; (case)
49      END;
      size 94

```

TASK TYPE rmt_task_type;

ENTRY get_resource(access access_type);

```

50      VAR
51      stuff : phy_actn;
52      BEGIN
53      CASE access OF
54      rd_access:
55      stuff := lgcl_to_phy_actn(prc_lgcl_evt(phy_to_lgcl_evt(p_get_rd_access)));
56      wr_access:
57      stuff := lgcl_to_phy_actn(prc_lgcl_evt(phy_to_lgcl_evt(p_get_wr_access)));
58      END; (case)
59      WHILE (stuff = p_accept_rls) DO
60      BEGIN
61      ACCEPT(rls_resource);
62      stuff := lgcl_to_phy_actn(prc_lgcl_evt(l_ri's_access));
63      END;
64      size 184

```

```

71      ENTRY rls_resource;
72      VAR
73      stuff : phy_actn;
74      BEGIN
75      stuff := lgcl_to_phy_actn(prc_lgcl_evt(l_ri's_access));
76      END;
77      size 62

```

```

78      BEGIN
79      WHILE true DO BEGIN
80      ACCEPT(get_resource);
81      rls_resource;
82      FND; (do forever)
83      END; (robot engine)
84      size 58

```

offset

size

85

86

87

88

89

90

91

1

1

1

1

1

1

0 BEGIN

-1 {

robot initialization

}

END.

0

size :

28

28

offset size

size 118

```

48      *
49      *
50      0
51      30
52      1
53      2
54      3
55      4
56      5
57      6
58      7
59      8
      -1
      size 94

function phy_to_lgcl_evtnt(evtnt : phy_evtnt) : lgcl_evtnt;
begin
  case evtnt of
    p_get_rd_access:
      phy_to_lgcl_evtnt := l_get_rd_access;
    p_get_wr_access:
      phy_to_lgcl_evtnt := l_get_wr_access;
    p_rls_access:
      phy_to_lgcl_evtnt := l_rls_access;
  end; {case}
end;

```

task type rm_task_type;

```

60
61
62
63
64
65
66      0
67      56
68      10
69      20
70      24
71      4
72      8
      size 122

entry get_write_access;
begin
  stuff := lgcl_to_phy_actnt(prc_lgcl_evtnt(phy_to_lgcl_evtnt(p_get_wr_access)));
  while (stuff = p_acpt_rls) do
    begin
      accept[rls_resource];
      stuff := lgcl_to_phy_actnt(prc_lgcl_evtnt(l_rls_access));
    end;
  end;
end;

```

```

73
74
75
76      0
77      54
      size 62

entry get_read_access;
begin
  stuff := lgcl_to_phy_actnt(prc_lgcl_evtnt(phy_to_lgcl_evtnt(p_get_rd_access)));
end;

```

```

78
79
80      2
81      0
82      54
83      8
      size 62

entry rls_resource;
var
  stuff : phy_actnt;
begin
  stuff := lgcl_to_phy_actnt(prc_lgcl_evtnt(l_rls_access));
end;

```

84

```

25 0
26 24
27 8
28 2
29 2
30 22
31 3
32 3
33 4
34 3
35 3
36 3
37 16
38 24
39 4
40 10
41 size
42 112
43
44 0
45 1
46 1
47 1
48 1
49 1
50 1
51 1
52 1
53 1
54 1
55 1
56 1
57 1
58 1
59 1
60 1
61 1
62 1
63 1
64 1
65 1
66 1
67 1
68 1
69 1
70 1
71 1
72 1
73 1
74 1
75 1
76 1
77 1
78 1
79 1
80 1
81 1
82 1
83 1
84 1
85 1
86 1
87 1
88 1
89 1
90 1
91 1
92 1
93 1
94 1
95 1
96 1
97 1
98 1
99 1
100 1
101 1
102 1
103 1

```

offset size

1 parallel unit rmtask;

2 interface

3 type

4 access_type = (rd_access,

5 wr_access);

6 wtr_rc = (ok,

7 wt);

8 task type rm_task_type;

9 provides

10 entry get_write_access :> (wrc : wtr_rc);

11 entry wt_write_access;

12 entry get_read_access;

13 entry rls_resource;

14 end;

15 implementation

16 uses acm;

17 type

18 phy_evt = (p_get_rd_access,

19 p_get_wr_access,

20 p_rls_access);

21 phy_actn = (p_nil_actn,

22 p_hlv_access,

23 p_allow_access,

24 p_wait_access);

25 var

26 stuff : phy_actn;

27 wtr_waiting : boolean;

28 function locl_to_phy_actn :> (actn : phy_actn)

29 begin

30 case actn of

31 p_get_rd_access :> p_get_rd_actn;

32 p_get_wr_access :> p_get_wr_actn;

33 p_rls_access :> p_rls_actn;

34 p_hlv_access :> p_hlv_actn;

35 p_allow_access :> p_allow_actn;

36 p_wait_access :> p_wait_actn;

37 end;

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125

126

127

128

129

130

131

132

133

134

135

136

137

138

139

140

141

142

143

144

145

146

147

148

149

150

151

152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

174

175

176

177

178

179

offset 576

```

48 48 0 2 1 lgcl_to_phy_actn = p_actn_access;
49 49 0 2 2 lgcl_to_phy_actn = p_actn_access;
50 50 10 2 3 lgcl_to_phy_actn = p_actn_access;
51 51 40 2 4 end {case}
52 52 10 2 5 end
size 178

```

```

53
54 *
55
56 0 1
57 30 2 1 case evnt of
58 30 8 2 p_getrd_access
59 30 8 2 phy_to_lgcl_evnt = l_getrd_access;
60 30 10 2 p_getwr_access
61 30 10 2 phy_to_lgcl_evnt = l_getwr_access;
62 48 10 2 p_rls_access
63 58 26 2 phy_to_lgcl_evnt = l_rls_access;
64 84 10 2 end; {case}
size 94

```

```

65
66
67 task_type rm_task_type;

```

```

68
69 entry get_write_access -> {wrc wtr_rc};
70
71 0 56 1 begin
72 56 8 2 stuff := lgcl_to_phy_actn(prc_lgcl_evnt(phy_to_lgcl_evnt(p_getwr_access)));
73 2 2 2 case stuff of
74 2 3 p_wait_access;
75 3 4 begin
76 64 10 3 wrc := wt;
77 74 8 3 wtr_waiting := true;
78 82 4 3 end;
79 86 10 2 p_blk_access;
80 96 28 2 wrc := ok;
81 124 8 1 end; {case}
size : 132

```

```

82
83 entry wt_write_access;
84 1 1 begin
85 1 1 {
86 1 1 No need to invoke the state machine here because
87 1 1 it was already done when the get_write_access
88 1 1 entry was accepted.

```


ADDRESS	TEXT	ADDRESS	TEXT	ADDRESS	TEXT	ADDRESS	TEXT
130	size	130	size	130	size	130	size
131	0	131	0	131	0	131	0
132	begin	132	begin	132	begin	132	begin
133	{	133	{	133	{	133	{
134	robot initialization	134	robot initialization	134	robot initialization	134	robot initialization
135	}	135	}	135	}	135	}
136	end	136	end	136	end	136	end
137	size	137	size	137	size	137	size
138	0	138	0	138	0	138	0
139	begin	139	begin	139	begin	139	begin
140	{	140	{	140	{	140	{
141	robot initialization	141	robot initialization	141	robot initialization	141	robot initialization
142	}	142	}	142	}	142	}
143	end	143	end	143	end	143	end
144	size	144	size	144	size	144	size
145	0	145	0	145	0	145	0
146	begin	146	begin	146	begin	146	begin
147	{	147	{	147	{	147	{
148	robot initialization	148	robot initialization	148	robot initialization	148	robot initialization
149	}	149	}	149	}	149	}
150	end	150	end	150	end	150	end
151	size	151	size	151	size	151	size
152	0	152	0	152	0	152	0
153	begin	153	begin	153	begin	153	begin
154	{	154	{	154	{	154	{
155	robot initialization	155	robot initialization	155	robot initialization	155	robot initialization
156	}	156	}	156	}	156	}
157	end	157	end	157	end	157	end
158	size	158	size	158	size	158	size
159	0	159	0	159	0	159	0
160	begin	160	begin	160	begin	160	begin
161	{	161	{	161	{	161	{
162	robot initialization	162	robot initialization	162	robot initialization	162	robot initialization
163	}	163	}	163	}	163	}
164	end	164	end	164	end	164	end
165	size	165	size	165	size	165	size
166	0	166	0	166	0	166	0
167	begin	167	begin	167	begin	167	begin
168	{	168	{	168	{	168	{
169	robot initialization	169	robot initialization	169	robot initialization	169	robot initialization
170	}	170	}	170	}	170	}
171	end	171	end	171	end	171	end
172	size	172	size	172	size	172	size
173	0	173	0	173	0	173	0
174	begin	174	begin	174	begin	174	begin
175	{	175	{	175	{	175	{
176	robot initialization	176	robot initialization	176	robot initialization	176	robot initialization
177	}	177	}	177	}	177	}
178	end	178	end	178	end	178	end
179	size	179	size	179	size	179	size
180	0	180	0	180	0	180	0
181	begin	181	begin	181	begin	181	begin
182	{	182	{	182	{	182	{
183	robot initialization	183	robot initialization	183	robot initialization	183	robot initialization
184	}	184	}	184	}	184	}
185	end	185	end	185	end	185	end
186	size	186	size	186	size	186	size
187	0	187	0				

END

03·03·92

FIN