An Implementation of the State Transition Event Model

by

Gordon Wayne O'Connell

B Sc , University of Victoria, 1976
B Sc , University of Victoria, 1991

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

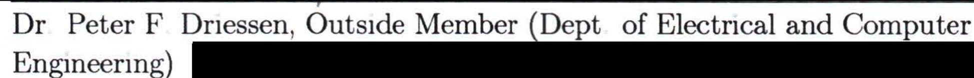
MASTER OF SCIENCE

in the Department of Computer Science

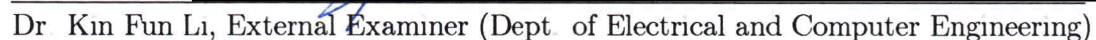
We accept this thesis as conforming to the required standard

Dr Gholamali C Shoja, Supervisor (Dept of Computer Science)

Dr Mantis Cheng, Departmental Member (Dept of Computer Science)



Dr Peter F Driessen, Outside Member (Dept of Electrical and Computer Engineering)



Dr Kin Fun Li, External Examiner (Dept of Electrical and Computer Engineering)

© Gordon Wayne O'Connell, 1997

University of Victoria

All rights reserved This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author

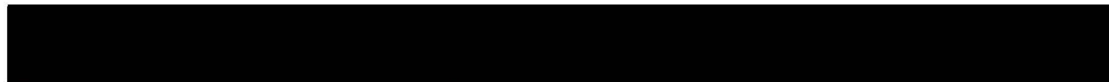
Supervisor Dr G C Shoja

Abstract

This thesis describes the design and implementation of a state transition event machine (STEM) A STEM is an abstract machine which combines the operations of a finite state machine with certain synchronization primitives in an attempt to control the execution of a collection of mutually cooperating tasks The declaration of a STEM can describe an abstract machine requiring hundreds of individual operations To address this problem, a precompiler was developed to convert STEM declarations into compilable implementations The synchronization requirements of a STEM are provided by low-level primitives called d-operations The state transition event machine generated by our precompiler can be verified by embedding the code into an event-driven simulator and analyzing the output of the simulation log

Examiners

Dr Gholamali C Shoja, Supervisor (Dept of Computer Science)



Dr Mantis Cheng, Departmental Member (Dept of Computer Science)

Dr Peter F Driessen, Outside Member (Dept of Electrical and Computer Engineering)

Dr Kin Fun Li, External Examiner (Dept of Electrical and Computer Engineering)

Table of Contents

Abstract ii

Table of Contents iii

List of Tables viii

List of Figures ix

List of Listings x

Acknowledgments xii

1 Introduction 1

1.1 State Transition Event Model 3

1.2 A Small Example 5

1.3 State Transition Event Operations 8

1.3.1 STE Operations 8

1.3.2 D-Operations 10

1.4 Objectives 11

1.5 Overview 12

2 Background 14

2.1 Synchronization Methods 15

2.2 State Transition Event Machine 19

3 Implementing High-Level Synchronization Operations 22

3.1 State Transition Event Machines 23

3.1.1 Example 1 Radar Set Module 24

3.1.2 Example 2 Mail Manager Module 26

3.1.3 Conditions and Events 28

3.1.4 Application Interface 29

3.2 Implementing a STEM to 'C' precompiler 31

3.2.1 Scanner 31

3.2.2 Parser 32

3.2.3 Semantic Analyzer 33

3.2.4 Code Generator 36

3.3 Code Generation Data Structures 36

3.3.1 STE Type Constants 36

3.3.2 Transition Matrix 38

3.3.3 Set Table 39

3.3.4 Condition Table 40

3.3.5 STE Variables 41

3.4 Code Generator STE Operations 42

3.4.1 STE Variable Constructor 45

3.4.2 STE Variable Destructor 46

3.4.3 STE Variable Update Operation 46

3.4.4 Set Inquiry Operation 47

3.4.5 Relation Inquiry Operation 47
3.4.6 State Transition Operation 48
3.4.7 Synchronization Operations 49
3.4.7.1 Wait on Transition Event 50
3.4.7.2 Wait on Call 50
3.4.7.3 Wait on Set Transition 51
3.4.7.4 Wait on Condition 53
3.4.7.5 Wait on Set Membership 54
3.5 STE Variables and the STE Model 56
3.6 Summary 56

4 Implementing Low-Level Synchronization Operations **58**
4.1 VSTa An Experimental Microkernel 58
4.2 STE D-Operations 60
4.2.1 Data Structures 64
4.2.2 Subsystem Initialization 66
4.2.3 dop_create and dop_destroy 66
4.2.4 dop_pass, dop_up and dop_down 70
4.2.5 dop_movesema 76
4.3 Transition Manager 78
4.3.1 Transition Manager Data Structures 80
4.3.2 Creating State Transition Event Machines 82
4.3.3 Destroying State Transition Event Machines 84
4.3.4 Updating State Transition Event Machines 85
4.4 Summary 88

5 Experience and Results **90**

5.1 Testing the STE Operations 90

5.1.1 Simulating the Radar Set Module 92

5.1.2 Test Threads 94

5.1.2.1 EVT Test Thread 95

5.1.2.2 CND Test Thread 96

5.1.2.3 MBR Test Thread 97

5.1.2.4 SET Test Thread 100

5.1.3 Simulation Results 103

5.2 Timing Characteristics of Synchronization Operations 108

5.2.1 dop_create and dop_destroy 109

5.2.2 Wait-Wake Response 110

5.2.3 dop_up and dop_movesema 113

5.2.4 High-Level Synchronization Operations 115

5.3 Summary 117

6 Conclusion **118**

6.1 Contributions and Achievements 119

6.2 Future Work 120

References **122**

A Sample STE Type Declarations **124**

A.1 Type Declaration for Radar Set Module 124

A.2 Type Declaration for Mail Manager Module 126

B Scanner and Parser Description Files **128**

B.1 Scanner Description File 128

B.2 Parser Description File 130

C Sample Precompiler Output 133

C.1 Interface for Radar Set Module 133

C.2 Implementation for Radar Set Module 137

D Programming Interfaces 144

D.1 STEM Operations 144

D.2 D-Operations 147

D.3 Transition Manager Operations 150

D.4 Set Operations 151

E Overview of Tool Set 153

E.1 stetoc - A STE to 'C' precompiler 153

E.2 verify - Testing Radar Set Module Operations 154

F State Machine Verification 157

F.1 Example Simulation Log 157

F.2 Verification Results 162

F.2.1 Verification results for canon 162

F.2.2 Verification results for canon20 162

F.2.3 Verification results for random10 163

F.2.4 Verification results for random20 164

List of Tables

1.1 Transition function for watchdog timer . . . 6

3.1 Transition function for radar set module . . . 25
3.2 Transition function for mail manager module . . . 27
3.3 Derivation of state set (K) for radar set module . . . 29
3.4 Warning and error messages generated by the semantic analyzer . . . 35
3.5 STE operations . . . 44

4.1 STE d-operations . . . 61
4.2 VSTa functions used in the design of STE d-operations . . . 63
4.3 Transition manager operations . . . 79
4.4 Transition manager: semaphore initialization . . . 84
4.5 Transition manager: semaphore update . . . 88

5.1 Simulation results . . . 104
5.2 Interfaces utilized by the test application . . . 105
5.3 Timing characteristics of STE operations . . . 116

List of Figures

1.1 State transition event model . . . 5

1.2 State transition diagram for watchdog timer . . . 7

3.1 State transition diagram for radar set module . . . 26

3.2 State transition diagram for mail manager module . . . 28

3.3 STE application structure . . . 30

3.4 Phases of the STEM to 'C' precompiler . . . 31

3.5 An abstract grammar for STE types declarations . . . 32

3.6 An implementation of a STE variable . . . 57

List of Listings

3.1 STE type constants for radar set module . . . 37
3.2 Transition matrix for radar set module . . . 38
3.3 Set table for radar set module . . . 40
3.4 Condition table for radar set module . . . 41
3.5 Radar STE variable . . . 42
3.6 Variable constructor `CreateRadar()` . . . 45
3.7 Variable destructor `DeleteRadar()` . . . 46
3.8 Variable update operation `UpdateRadar()` . . . 47
3.9 Set inquiry operation `IsTracking()` . . . 47
3.10 Relation inquiry operation `IsTrack()` . . . 48
3.11 State transition operation `Track()` . . . 48
3.12 Wait on transition event synchronization operation `awaitEvtTrack()` 50
3.13 Wait on call synchronization operation `awaitCallTrack()` . . . 51
3.14 Wait on set transition synchronization operation `awaitSettTrack()` 52
3.15 Wait on condition synchronization operation `awaitCndtTrack()` and `awaitCndfTrack()` . . . 54
3.16 Wait on set membership synchronization operation `awaitMbrtTracking()` and `awaitMbrfTracking` . . . 55
4.1 Data structures: VSTa spinlocks and semaphores . . . 65

4.2 STE semaphores subsystem initialization 66
4.3 Create STE semaphore operation dop_create() 68
4.4 Destroy STE semaphore operation dop_destroy() 69
4.5 Create STE semaphore utility functions 70
4.6 Semaphore passage operation dop_pass() 72
4.7 Semaphore opening operation dop_up() 74
4.8 Semaphore closing operation dop_down() 75
4.9 Semaphore morphing operation dop_movesema() 78
4.10 Data structures state transition event machine 81
4.11 Create state transition event machine 83
4.12 Update state transition event machine 87
5.1 EvtTrack a thread to test awaitEvtTrack 96
5.2 CndTrack a thread to test awaitCndTrack 98
5.3 MbrfMode a thread to test awaitMbrtMode 99
5.4 SetTracking a thread to test awaitSetTracking 101
5.5 Set tests for awaitSetTracking 102
5.6 Measuring the time to execute an operation 109
5.7 Measuring wait-wake response 112
5.8 Measuring median restart time 113
5.9 Measuring median morphing time 114

Acknowledgements

I wish to thank my supervisor, Dr. G. C. Shoja for his advise and encouragement throughout this thesis. Dr. G. C. Shoja suggested this research project, and I am grateful for his guidance and support as the research progressed.

I would like to thank Dr. M. Cheng for his constructive criticism of my research project and for his advise on how to describe the STE model.

I have benefited greatly from the diverse approaches to research suggested by my course instructors during my M. Sc. program at the University of Victoria.

I would like to thank Dr. David Parnas at McMaster University, and, Dr. Stuart Faulk at the University of Oregon, for demonstrating how state transition events can be used to structure systems.

Finally, I would like to thank my wife, Pat, for her support, tolerance, and just for being there.

Chapter 1

Introduction

This thesis describes our implementation of a state transition event machine (STEM), an abstract data type which encapsulates a deterministic finite automaton (DFA) and synchronization objects which react to state transition events in the underlying DFA. The work described here is related to the finite state machine model [1]. We believe that despite certain shortcomings, such as the state explosion problem, the finite state machine model can be a very powerful and appropriate tool for designing systems that can be easily understood, verified and maintained. The approach to finite state machine modeling followed here was introduced by Faulk and Parnas [2, 3, 4]. These authors introduce the *state transition event model* and discuss its main advantages:

- STE variables[1] provide an appropriate abstraction which allows direct modeling of certain classes of hard-real-time problems, such as embedded computer systems.

[1]STE = state transition event. A STE variable is an instance of some STEM data type.

- STE variables permit separation of concerns in that,
  - processes do not signal specified processes, they utilize access operations on commonly defined STE types
  - waiting processes do not wait for signals from specified processes, they wait for certain state changes in an STE variable
- STE variables allow an efficient implementation

The experimental operating system used in this work supports a simple threads model allowing multiple concurrent threads to share a common address space. The synchronization facility we describe permits threads to cooperate in accessing a shared address space. To provide the synchronization support suggested by Faulk and Parnas [3], it was necessary to extend the system call interface of our experimental microkernel.[2] We provided these synchronization services through an implementation of the *d-operations* described by Belpaire and Wilmotte [5].

Using a published specification for state transition event machines, we examine one implementation and provide a compilation tool which can convert a STEM declaration into compilable 'C' code. This implementation of STEMs is provided as a programming facility for systems that can be represented as event-driven state machines.

To verify that our implementation satisfies the specification in [3], we designed a simulator for the *Radar Set Module*. The simulator inputs event traces and outputs a log of all transition and synchronization events. We analyze the simulation output with a simple verification script.

[2] VSTa release 1.5.2

1.1 State Transition Event Model

The *state transition event model*, or STE *model*, is an approach for designing event-driven systems as a set of cooperating sequential processes. The STE *model* addresses the problem of execution sequencing. Sequencing is a mechanism that allows an application to order the operations in different processes to achieve co-operation in performing a common task. We illustrate this model in Figure 1.1. A system designed using this model is composed of the following objects:

- a deterministic finite automaton (DFA), labeled $\mathcal{M}$,
- event-detecting processes, like P_a and P_b, which recognize the input language of the DFA,
- a condition table, (a condition is a predicate that characterizes some aspect of the system state for a measurable period of time), and,
- condition-enabled processes, like P_x and P_y, which wait for specific conditions in the condition table to become valid.

In Figure 1.1 for example, processes P_a and P_b detect and signal the occurrence of events a (an external event) and b (an internal event). The signals cause state transitions within the underlying state machine ($\mathcal{M}$), which force the update of entries in the condition table. Processes P_x and P_y are initially blocked on semaphores associated with the condition table. Changes in the condition table can cause processes P_x and P_y to be restarted. In this example, condition α restarts process P_x, which generates the output O_α, while β restarts process P_y, which performs some unspecified internal operation. We can describe this system in terms of its input

and the resulting abstract computation. For the system in Figure 1.1 for example, the following computation is possible.

$$\text{input trace} : abbaab$$

$$\text{abstract computation} : a\alpha b\beta b\beta a\alpha a\alpha b\beta$$

where (a,b) are symbols from the input language of the DFA, $\mathcal{M}$, and (α,β) identify the conditions which became valid during the computation.

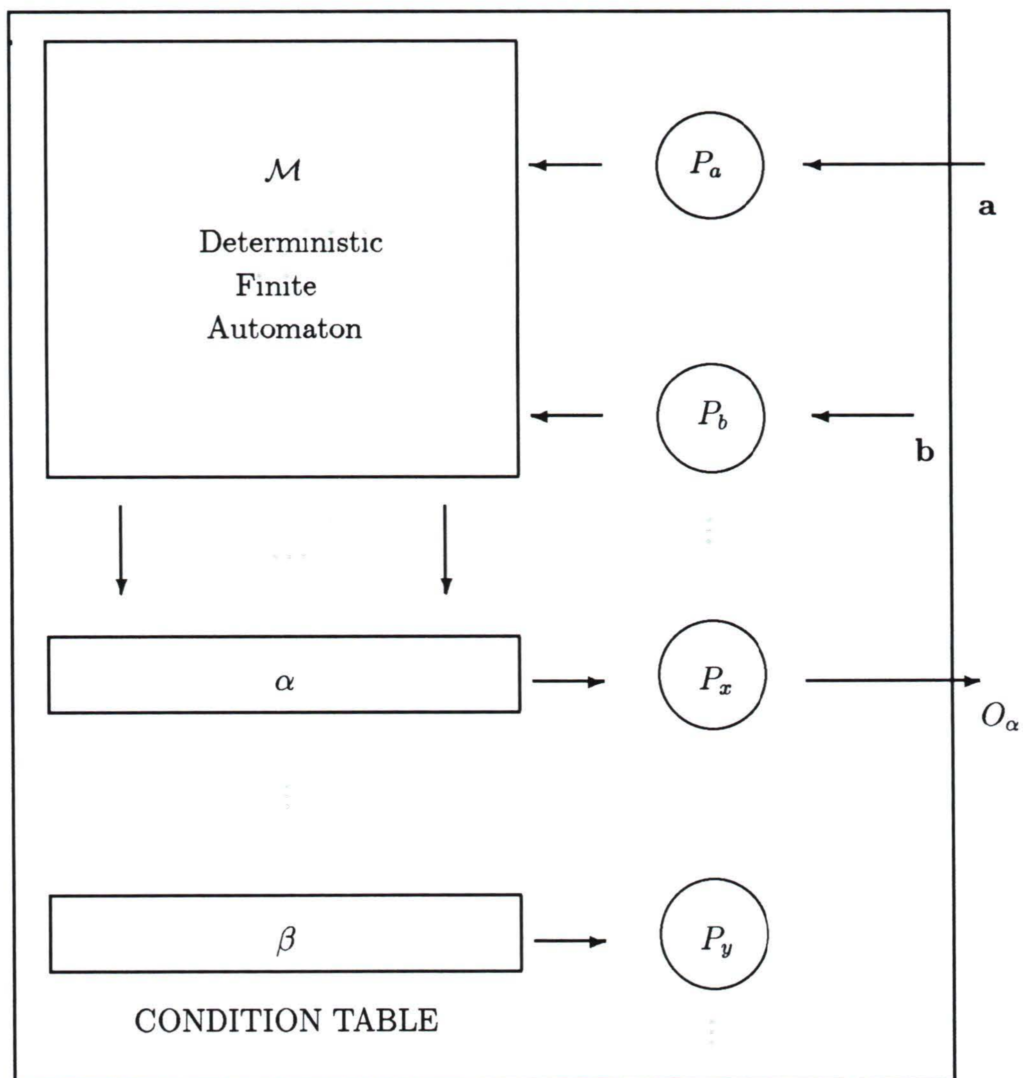


Figure 1 1 State transition event model

1.2 A Small Example

In the STE *model*, information about system state and state changes are captured using a deterministic finite automaton (DFA). In this section we show how a formal

representation of a system, expressed as a DFA, can be translated into a STE type declaration In Chapter 3 we describe a tool which can translate a specification, given as a STE type declaration, into an implementation which includes all of the passive components in the STE *model*, namely, $\mathcal{M}$ the deterministic finite automaton, the condition table and the required synchronization structures We call such an implementation a state transition event machine (STEM)

Watchdog timers are sometimes used to monitor the progress of an application Once enabled, a watchdog timer must be reset periodically If not reset, the timer expires and an exception is raised [6] A state transition diagram for a watchdog timer is given in Figure 1 2

From this diagram we can construct the following DFA, $\mathcal{M} = (K, \Sigma, s, F, \delta)$ where

K = {WAIT,ENABLED,HALT}, a finite set of states,

Σ = { e, r, t }, the input alphabet (events),

s = WAIT, the initial state,

F = {HALT}, the set of final states

and δ, the transition function, is given in Table 1 1

Current State	**Event**[3]		
	e	r	t
WAIT	ENABLED	X	X
ENABLED	X	ENABLED	HALT
HALT	X	X	X

Table 1 1 Transition function for watchdog timer

[3]Events e = enable, r = reset, t = timeout

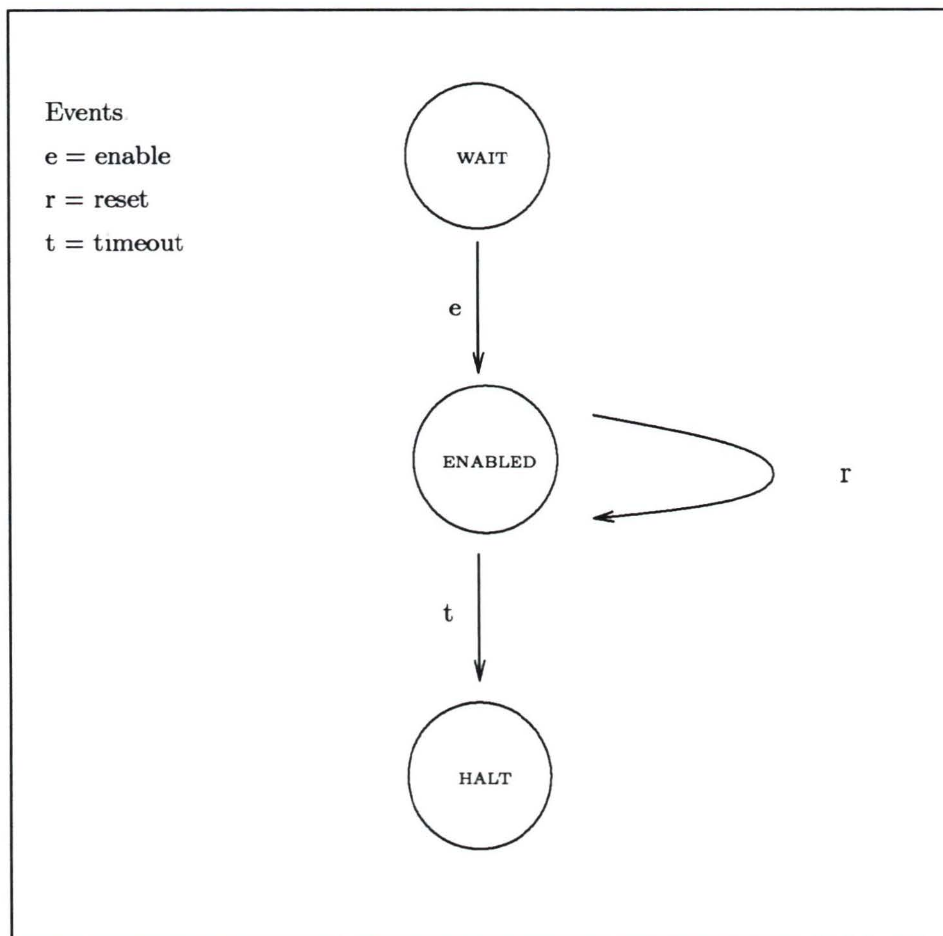


Figure 1 2 State transition diagram for watchdog timer

We can translate this formal specification of a watchdog timer into the following STE type declaration

```
1    ste watchdog
2    {
3        state { wait, enabled, halt },
4
5        relation enable { (wait->enabled) },
6
7        relation reset { (enabled->enabled) },
8
9        relation timeout { (enabled->halt) },
```

```
10
11   },
```

This declaration describes the watchdog STE type. The states in the formal specification are simply listed as elements in the declaration's *state attribute*. Each event in the input language of the formal specification is converted to a distinct *relation attribute*. Each *relation attribute* contains one or more of the transitions from the original specification. STE declarations can also include *set attributes*, which describe a partitioning of the DFA's state space. Set attributes have not been declared for the watchdog timer application. Notice that the transition function, δ, is not included in the declaration. The precompiler described in Chapter 3 can construct δ from the information in the *state* and *relation attributes*. We also do not translate information concerning the initial state (s) and the set of final states (F). These are handled programatically.

1.3 State Transition Event Operations

In the STE *model*, event-signaling is supported by a synchronization mechanism referred to as *state transition event synchronization* [3]. STE synchronization is provided by operations on a type of variable called a STE variable. A STE variable is an instance of a STE type declaration. Two classes of operations are required for the implementation of STE variables: STE *operations* and *d-operations*.

1.3.1 STE Operations

In the following, <u>Type</u> indicates the name of the STE declaration, <u>Set</u> represents a *set attribute*, <u>Relation</u> represents a *relation attribute* and p, q are STE variables.

Given a STE type declaration, the available STE operations include:

Create*Type*(p), STE variable constructor: Constructor for STE variables of type *Type*.

Delete*Type*(p), STE variable destructor: Destructor of STE variables of type *Type*.

Update*Type*(p,event), STE variable updater: Used by event-signaling processes to update the STE variable p. The event parameter must belong to the input alphabet of the underlying DFA.

Is*Set*(p), set inquiry operation: A predicate operation which tests whether $S \in T$, where S is the current state of STE variable p, and T is the set *Set*.

Is*Relation*(p,q), relation inquiry operation: A predicate operation which tests whether $(S \rightarrow S') \in R$, where S is the current state of STE variable p, S' is the current state of STE variable q, and R is the relation *Relation*.

***Relation*(p), state transition operation:** These operations generate state changes in the underlying DFA. They perform the next state calculation subject to the constraint $(S \rightarrow S') \in R$, where S is the current state of the DFA, S' is the next state of the DFA and R is the relation *Relation*. The *state transition operations* are implemented using the STE *variable update operation*, Update*Type*().

awaitCall*Relation*(p), wait on call operation: Processes executing this operation are suspended until an event-detecting process uses the *state transition operation* associated with the relation *Relation*.

awaitEvt*Relation*(p), wait on transition event operation: Processes executing this operation are suspended until an event causing a transition belonging to relation *Relation* occurs. Superficially, the *wait on transition event operations* are identical to the associated *wait on call operations*. The specification of STE variables given in Faulk [4] describes the conditions where *wait on transition event operations* are required.

awaitSet*Set*(p), wait on set transition operation: Processes executing this operation are suspended until the system state changes from a state *outside* of the set *Set* to a state *inside* of the set *Set*.

awaitCnd*Relation*(p,q), wait on condition operation: Processes executing this operation will block if the condition $(S \rightarrow S') \in R$ does not hold, where S is the current state of STE variable p, S' is the current state of STE variable q, and R is the relation *Relation*. This condition is satisfied when state S' becomes reachable from the current state S in a single transition.

awaitMbr*Set*(p), wait on set membership operation: Processes executing this operation will block if the current state of STE variable p is not a member of the set *Set*.

Our implementation of the STE operations is presented in Section 3.4.

1.3.2 D-Operations

The synchronization required so that event-signaling processes can restart condition-enabled processes is provided by the *d-operations* [5]. *D-operations* are based on three synchronization primitives:

`pass()`, semaphore passage: The semaphore passage operation allows a process to suspend execution. When a process executes the passage operation and the value of the semaphore is negative, the operation is not completed and execution of the process is suspended until the semaphore becomes nonnegative.

`down()`, semaphore closing: The semaphore closing operation decrements the value of the semaphore. Execution of the closing operation can never suspend the executing process.

`up()`, semaphore opening: The semaphore opening operation increments the value of the semaphore. When the value of the semaphore becomes nonnegative, *all processes* suspended on the semaphore are restarted.

The *d-operations* are quite different from the `P` and `V` operations described by Dijkstra[7]. Atomically combining the `pass()` and `down()` operations (denoted `pass(); down()`), leads to the `P` operation. The `up()` operation is similar to the `V` operation, except that `up()` is a *broadcast* primitive, restarting all suspended processes on a semaphore while `V` is best described as a *signal* primitive, releasing at most one process per invocation.

Our implementation of the *d-operations* is presented in Section 4.2.

1.4 Objectives

The specification for STEMs has been available for some time [2, 3, 4]. The goal of this thesis is to examine one possible implementation of STEMs. To achieve this objective we have concentrated our research in the following areas:

- Develop an automation tool which can convert STEM declarations into compilable 'C' code. A STEM declaration can imply hundreds of operations. This tool will allow programmers to design their applications without the necessity of coding these state machine operations directly.

- Provide synchronization support for STEMs.

- Develop a technique to verify the operations implied by a STEM.

- Investigate certain aspects of the execution performance of our implementation.

1.5 Overview

In Chapter 2, we present the works that have influenced our design and implementation of STEMs. We examine approaches to synchronization and the primary literature on STEMs.

In Chapter 3, we discuss in more detail what a STEM is. We describe the structure of our automation tool for converting STEM declarations into compilable 'C' code. This is followed by a description of how each STEM operation is realized.

In Chapter 4, we discuss how we provide synchronization support for STEMs. We also describe how run-time updating of the state machine is accomplished.

In Chapter 5, we describe the techniques used to verify a running STEM. We describe how timing measurements were taken and discuss aspects of the performance of our implementation of STEMs.

In Chapter 6, we state our findings and contributions.

In Appendix A, we provide two examples of STEM type declarations.

In Appendix B, we provide the description files for the scanner and parser components of our automation tool.

In Appendix C, we provide the interface and implementation modules generated by our automation tool, for the Radar Set Module.

In Appendix D, we provide the programming interfaces for the services required by STEMs. These include interfaces for the synchronization operations, the transition manager, and, general set operations.

In Appendix E, we describe the tool set developed to work with STEMs.

In Appendix F, we provide an extract of a simulation log, and the verification results for a number of simulator runs.

Chapter 2

Background

An approach used in computer science to describe and analyze a system is to develop an abstract machine, a mathematical formalism in which the concepts of state and transition between states are precisely defined [1]. Probably the most common example of an abstract machine in computer science is the *state machine*.

Heitmeyer and Mandroli [1] summarize the role of *finite state machines* in the design of real-time systems. *Finite state machines* are often used in the specification of real-time systems. Specifications based on *finite state machines* are called operational specifications because they describe the systems operating rules. Heitmeyer and Mandroli list some of the limitations of the classical state machine in model specification.

- Timing-dependent behaviour cannot be expressed.
- Captures only a small fraction of system behaviour.

2.1 Synchronization Methods

An excellent introduction to concurrent programming can be found in [8]. Two important uses for synchronization mechanisms are *mutual exclusion* and *general waiting* [9]. The commonest mechanisms employed to achieve mutual exclusion are *mutexes* and *semaphores* [8, 9, 10]. Both mechanisms are often implemented using the P and V synchronization primitives described by Dijkstra [7]. A *mutex* abstract data type consists of a *mutex* synchronization object (lock and state variables) and two operations, `acquire()` which requests exclusive access to a synchronization object, and `release()` which relinquishes exclusive access of a synchronization object. A *semaphore* may be viewed as a Boolean variable corresponding to a condition which must be satisfied before a thread can proceed [8]. The *semaphore* abstract data type consists of an integer and two operations, `wait()` which blocks a thread when the condition is false, and `signal()` which is used to restart a thread when a condition becomes true. The most serious problems for threads using these synchronization objects are *deadlock* and *priority inversion*. *Deadlock* can occur when all active threads in a system are blocked on synchronization objects, or when two or more threads compete to acquire a small collection of synchronization objects. *Priority inversion* arises when a high-priority thread is blocked on a synchronization object 'owned' by a low-priority thread. A problem arises when threads of intermediate priority, with no synchronization requirements, are restarted and block the low-priority thread. As an aid in solving this problem mutexes often have an additional property, the *ownership property*, the thread releasing a mutex must be the same thread which acquired the lock. If an operating system recognizes the *ownership property* of synchronization objects in advance, certain optimizations

can be realized. One such optimization is the *priority inheritance* protocol which can be used to guard against priority inversions [9]. In semaphores signaling is always anonymous.

The most common mechanism employed for synchronization involving waiting is the condition variable [11, 8, 10]. Condition variables are an implementation of the *conditional critical region* abstract data type first proposed by Brinch Hansen [12, 8]. A condition variable is an abstract data type which includes: a condition object with operations `wait()`, `signal()`, and `broadcast()`, some state variables, and, a predicate which can be applied to the state variables to determine the current condition. A condition variable is always associated with a *mutex* which protects the state variables. A thread accesses the condition variable by initially acquiring the *mutex lock*. The thread can then evaluate the predicate. If the predicate holds, the thread can access the critical region abstracted by the condition variable. If the predicate fails however, the thread executes the `wait()` operation and is suspended until some other thread re-evaluates the predicate. The `wait()` operation always releases the lock to the *mutex* associated with a condition variable. When the condition becomes true a thread may restart waiting threads using either the `signal()` operation or the `broadcast()` operation. `signal()` allows one thread to proceed, `broadcast()` restarts all threads blocked on a condition queue. Since this synchronization object contains a *mutex* it suffers the problems of *deadlock* and *priority inversion*. An additional problem is the *intercepted wakeup problem* [10]. Simply put, there are no guarantees that the predicate condition holds after signaled threads have been restarted. In the *broadcast* situation for example, a thread leaving the condition queue is free to alter any of the state variables. In this

situation conditions may not hold. The recommended solution for this problem is to have all restarted threads re-evaluate the predicate and determine if they should proceed or wait again. This gives rise to two potential problems:

starvation: Threads that are required to re-evaluate predicates may not be able to make progress.

loss of events: Some threads may only be interested in the frequency of events, not the value of a predicate. Such threads will have the 'impression' that events are lost.

Not all synchronization operations are based on the `P()` and `V()` primitives introduced by Dijkstra [7]. Belpaire and Willmotte [5] have described the *d-operations* which we have introduced in Section 1.3.2.

Birrel *et al.* [13] have described the implementation of *mutexes*, *semaphores*, and *condition variables* in the Taos operating system. They provide a formal language specification for semaphores, mutexes and condition variables using the *Larch Shared Language*. In their specification of mutexes the authors correctly include the *ownership property*. In a correct implementation of mutexes only the thread which *acquires* a mutex may *release* it. This is not the case for semaphores where the thread *posting* the semaphore is completely anonymous. In their specification of condition variables, Birrel *et al.* [13] describe two implementation problems. The first problem they encountered is the *intercepted wakeup problem* [10]. In a concurrent system, multiple threads may be blocked on the same condition variable. After the condition variable has been signaled however, the first awakened thread will have guarded access to the state variables. This can invalidate the original

predicate. Threads subsequently released from the condition queue should always re-evaluate the predicate before deciding whether to proceed. The second problem identified by Birrel *et al.* [13] is the *wakeup-waiting race* (referred to as *spurious wakeup* in [10]). This condition occurs when the operations performed by the *wait* primitive are not atomic relative to the execution of the *signal* primitive. The *wait* primitive must leave the critical section *and* suspend the executing thread in one atomic action. Signals can be lost and waiting threads may deadlock if *signal* primitives are allowed to interrupt the execution of the *wait* primitive. Birrel *et al.* [13] implemented synchronization primitives for the Taos operating system using their formal specifications. Their experience with the Taos thread model suggests that formal specifications of concurrent programs can be used productively by systems programmers.

LaMarca [14] investigates the performance of lock-free protocols that provide synchronization on shared-memory multiprocessors. Lock-based synchronization protocols are not well suited to asynchronous systems. The delay of a thread holding a lock can severely degrade a systems performance. This can lead to the following problems:

deadlock: All active threads in the system are waiting on locks.

priority inversions: A high-priority thread is unable to make progress because a low-priority thread holds a required lock.

convoying. Where the parallelism in a system cannot be exploited because threads move together from lock to lock.

Synchronization protocols which have the non-blocking property are generally not

subject to starvation, the system is unable to prevent some threads from making progress. A number of software protocols have been developed which generate lock-free concurrent objects. However, when compared to lock-based protocols such as spin locks [15], lock-free synchronization protocols (LFSP) have higher latency and generate more contention. LaMarca [14] cites three architectural characteristics of shared-memory multiprocessor systems which affect the performance of LFSPs:

- Remote memory accesses are more costly.
- Hardware synchronization primitives, like *Test&Set* and *Compare&Swap*, are more expensive instructions when compared to normal memory accesses.
- Lock-free protocols often behave optimistically, allowing multiple threads to proceed with an update as if they will succeed. This consumes systems communications bandwidth and interferes with the progress of successful threads.

LaMarca [14] develops a performance model for LFSPs and applies it to existing protocols. This results in an improved protocol, *solo-cooperation protocol*, which offers the same performance as spin locks and an added tolerance to delays by negating the affects due to optimistic synchronization policies. Threads implementing *solo-cooperation protocol* are immune from deadlock but susceptible to livelock.

2.2 State Transition Event Machine

The state transition event machine (STEM) has been introduced in Sections 1.1–1.3. In this section we provide additional background on STEMs.

In their STE model, Faulk and Parnas [3] describe a language of events and conditions. This language allows the programmer to write in terms of transitions in the abstract machine. This high-level abstraction utilizes a *finite state machine*. The bulk of their paper is an operational specification for a state transition event machine (STEM), an abstract state machine that utilizes synchronization primitives to respond to changes in the underlying state machine. In STEMs all information is encoded in the logic of the abstract state machine and programs are concerned only with the state of a variable, not the state of other processes. Processes are only delayed for the time it takes to update one STE variable. The STEM provides the facility needed to handle events occurring in real-time [3].

Faulk [4] uses state transition events as a formal mechanism for specifying event-driven real-time systems. Faulk describes the steps necessary to convert a translation table (essentially a list of events and conditions) into a mode transition table. He shows that fundamentally different mode transition tables will be generated if the user does not carefully distinguish between *events that cannot occur* and *events which do not change*. Other researchers [16] have shown that the mode transition table can be easily transformed into a finite structure suitable for model checkers. Faulk [4] also provides several examples of how complex events can be handled with the STE method. Complex events have the form:

```
@T(condition1) when (condition2)
```

which indicates that the event associated with `condition1` can only occur while `condition2` holds.

Faulk [4] feels that the *d-operations* meet all of the synchronization requirements for the STE model:

Exclusion regions: There is an efficient method for translating the specifications for an exclusion region into a program written in terms of *d-operations*.

Synchronization operations: *D-operations* can be used to constrain the sequencing of execution in different processes. This allows the programmer to order operations in different processes to cooperate in performing the task at hand.

Chapter 3

Implementing High-Level Synchronization Operations

Modeling real-time systems using state transition events (STE) requires three principal tools:

- a precompiler to convert STE type declarations into compilable code,
- a transition manager to manage state transition events, and,
- an executive to provide run-time services (thread management, scheduling, synchronization).

In the remainder of this chapter we will discuss the implementation of a STEM precompiler, `stetoc`, developed to convert STE type declarations into compilable 'C' code. Such a tool is useful for the following reasons:

- It automatically generates a state transition event machine (STEM) and the STE operations implied by a STEM declaration,

- It reports on the requirements for system resources (storage requirements for the state transition event machine, number of semaphores),
- It allows programmers to design applications with complex synchronization requirements using high-level synchronization constructs.

3.1 State Transition Event Machines

Defintion 3.1 (State Transition Event Machine) *A state transition event machine (STEM) is an abstract data type which encapsulates an enhanced deterministic finite automaton. A STEM can associate synchronization operations with any of the DFA's transitions. In addition to the DFA a STEM encapsulates a representation for sets of states referred to as modes. Modes can also be associated with synchronization operations. Other operations associated with a STEM include state inquiry operations and state transition operations.*

The types of operations which must be available to utilize state transition events in real-time systems have been discussed in [3, 4]. Given a STE declaration (see the examples in Appendix A.1 and A.2), the numbers and types of STE operations are easily enumerated.

The syntax and semantics of STE variables have been presented in [2, 4]. One of the objectives of our current research is to examine a representation of state transition event machines and discuss how this representation can be used to realize the STE operations. Every STEM is represented as a deterministic finite automaton.

Defintion 3.2 (Deterministic Finite Automaton [17]) *A deterministic finite automaton (*DFA*) is a quintuple* $M = (K, \Sigma, \delta, s, F)$ *where,* K *is a finite set of states,* Σ *is an alphabet,* $s \in K$ *is the initial state,* $F \subseteq K$ *is the set of final states and* δ*, the transition function, is a function from* $K \times \Sigma \rightarrow K$.

Each STE type is a family of finite state machines. A STE variable is an instance of a finite state machine whose states are the values of the variable. A STE type is created by defining a *set of states* (K) for the finite state machine and a *set of relations*. A relation $a \in \Sigma$ is defined by $\{(s,t) : \delta(s,a) = t \text{ for all } s,t \in K\}$. In the next two sections we provide examples of state transition event machines.

3.1.1 Example 1: Radar Set Module

The radar set module was first described in [3, 4]. The radar set module is a software module in the AE-7 avionics system which implements a virtual radar device. The radar device has three modes of operation designated ranging, tracking, and standby. These modes are mutually exclusive. The characteristics of the radar inputs are such that ranging mode must be used during target acquisition and tracking mode used during air-to-air combat, either mode may be used during normal flight operation. In standby mode the radar performs only self-checking operations.

Under certain conditions, usually temporary, data from the radar become unreliable. This condition can be detected by the software and may occur in any of the three modes. In addition, radar hardware may fail completely. Once a failure occurs, the radar never resumes operation. A STE type declaration for the radar set module is given in Appendix A.1. The observable states and state transitions

of the radar system are depicted in Figure 3.1.

Figure 3.1 describes a Radar Set Module DFA, $M = (K, \Sigma, s, F, \delta)$ where

$K = \{$TRKREL,TRKNOT,RNGREL,RNGNOT,STBYNOT,STBYREL,FAILED$\}$,

$\Sigma = \{$ t, r, s, x, x̃, f $\}$,

$s =$ STBYNOT,

$F = \{$FAILED$\}$

and δ, the transition function, is given in Table 3.1.

Current State	Event[1]					
	t	r	s	x	$\bar{x}$	f
TRKREL	X	RNGREL	STBYREL	X	TRKNOT	FAILED
RNGREL	TRKREL	X	STBYREL	X	RNGNOT	FAILED
STBYREL	TRKREL	RNGREL	X	X	STBYNOT	FAILED
TRKNOT	X	RNGNOT	STBYNOT	TRKREL	X	FAILED
RNGNOT	TRKNOT	X	STBYNOT	RNGREL	X	FAILED
STBYNOT	TRKNOT	RNGNOT	X	STBYREL	X	FAILED

Table 3.1: Transition function for radar set module.

[1]Events: t = track, r = range, s = standby, x = reliable, $\bar{x}$ = unreliable, and f = failure.

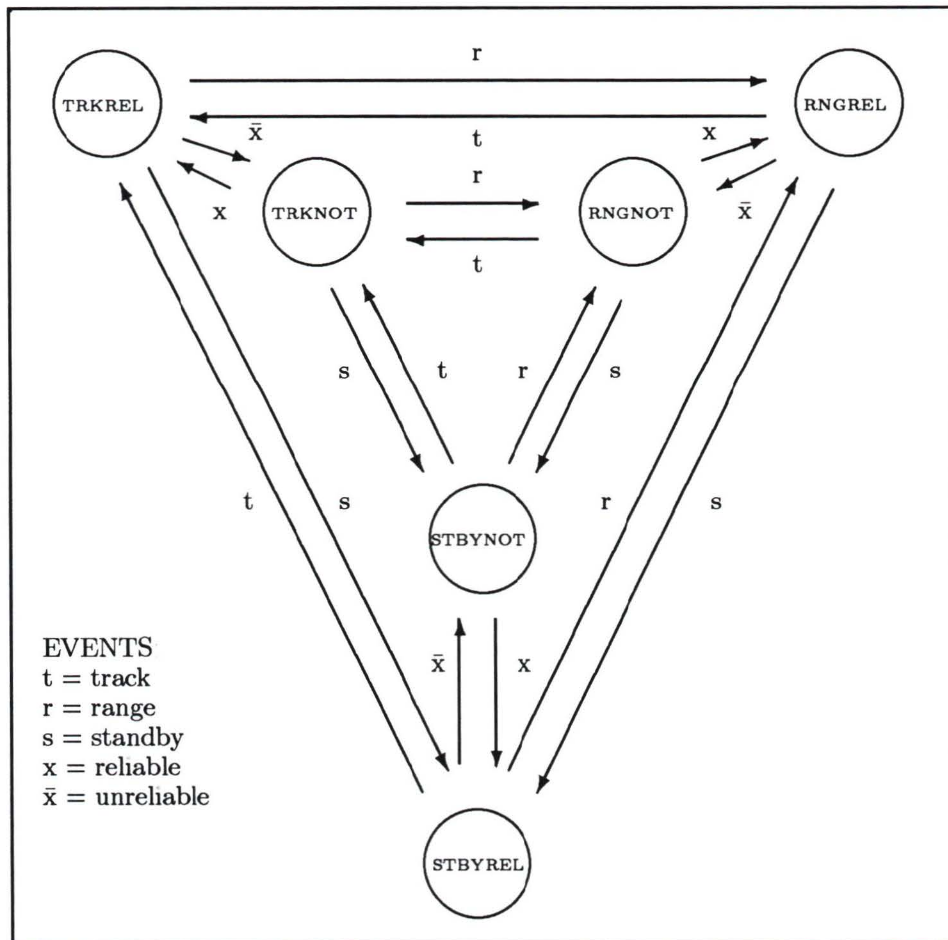


Figure 3 1 State transition diagram for radar set module

3.1.2 Example 2: Mail Manager Module

The mail manager module was first described in [2] The mail manager module controls a multi-host email system Users on any host may send email to users at any other site To reduce system load, hosts will try to batch outgoing messages Hosts will normally only deliver email at certain times and only if there is email waiting when the time to call occurs If there is no email to forward at a host site,

the host waits until the next call period. Users of the host computer can force the system to process email by requesting a call. If a call is not already in progress and the system is operating, a call is initiated as soon as possible. The mail manager monitors itself and the communications system for failures. Once a failure occurs, the mail manager suspends transmissions and informs the system operation. A STE type declaration for the mail manager module is given in Appendix A.2. The observable states and state transitions of the mail system are depicted in Figure 3.2. Figure 3.2 describes a Mail Manager DFA, $M = (K, \Sigma, s, F, \delta)$ where

$K = \{$RDYNIL,RDYMSG,CALLING,CALLREQ,NRDYNIL,NRDYMSG,FAILED$\}$,

$\Sigma = \{$ e, m, r, d, c, f $\}$,

$s =$ NRDYNIL,

$F = \{$CALLING,FAILED$\}$

and δ, the transition function, is given in Table 3.2.

Current State	**Event**[2]					
	e	m	r	d	c	f
RDYNIL	X	RDYMSG	CALLREQ	NRDYNIL	X	FAILED
NRDYNIL	RDYNIL	NRDYMSG	CALLREQ	X	X	FAILED
RDYMSG	X	RDYMSG	CALLREQ	NRDYMSG	CALLING	FAILED
NRDYMSG	RDYMSG	NRDYMSG	CALLREQ	X	X	FAILED
CALLREQ	X	X	CALLREQ	X	CALLING	FAILED
CALLING	X	X	X	X	X	FAILED

Table 3.2: Transition function for mail manager module.

[2]Events: e = enable, m = msgarrival, r = requestcall, d = disable, c = callinit, and f = failure.

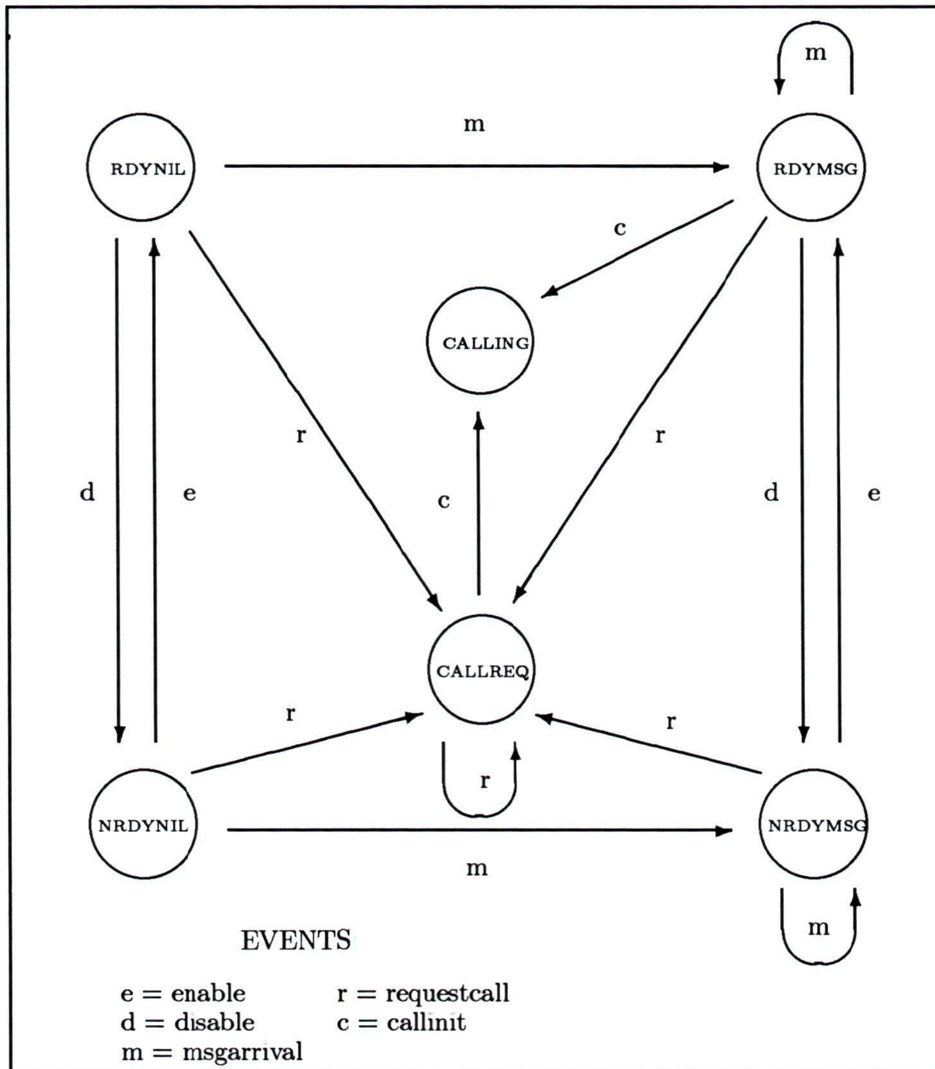


Figure 3 2 State transition diagram for mail manager module

3.1.3 Conditions and Events

Information characterizing state and state change can be described using a language of conditions and events A *condition* is a predicate that characterizes the system state for a measurable period of time The change in the value of a condition is

called an *event*. For example, using a notation introduced in [3], we can describe the *flight mode* of the Radar Set Module using one of the following *conditions*: `fmode=Standby`, `fmode=Ranging`, `fmode=Tracking`. In a similar fashion we can describe the *reliability mode* of the Radar Set Module using one of the following *conditions*: `rmode=Reliable`, `rmode=Unreliable`. We construct the set of states (K) for the Radar Set Module STEM by forming the Cartesian product **fmode** $\times$ **rmode**, as shown in Figure 3.3.

rmode	**fmode**		
	Standby	Ranging	Tracking
Reliable	STBYREL	RNGREL	TRKREL
Unreliable	STBYNOT	RNGNOT	TRKNOT

Table 3.3: Derivation of state set (K) for radar set module.

To describe the event of a condition becoming *true* we use the notation `@T(condition)`. The event of a condition becoming *false* is described as `@F(condition)`. As an example, in Figure 3.1 the transition (TRKREL, RNGREL) $\in$ **Range** generates two events: `@T(fmode=Ranging)` and `@F(fmode=Tracking)`.

3.1.4 Application Interface

Applications using STE variables require access to three programming interfaces. These are shown in Figure 3.3:

- STE *operations*: These are the high-level synchronization operations. They are generated by the `stetoc` precompiler based on a STE type declaration. The implementation of STE operations is discussed in Sections 3.3–3.4.

- *d-operations* These are the low-level synchronization operations These operations are a specialization of the counting semaphores provided by our run-time executive The implementation of d-operations is discussed in Section 4 2

- *transition manager* The transition manager is responsible for creating, destroying and updating STE variables STE variables are discussed in Sections 1 3, 3 3 5 and 3 5 The implementation of a transition manager is discussed in Section 4 3

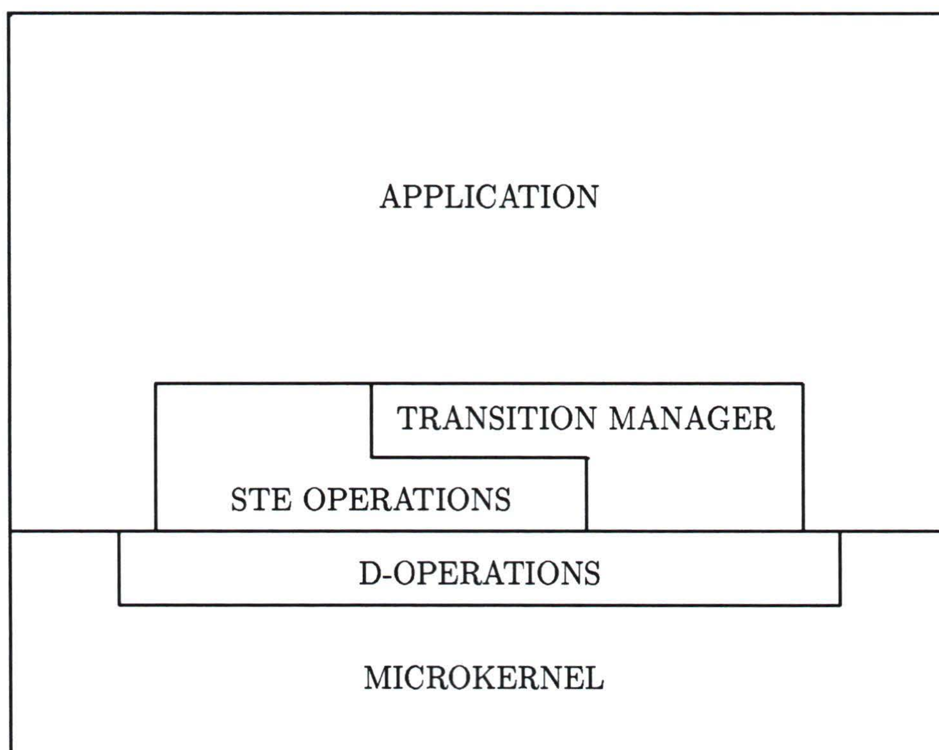


Figure 3 3 STE application structure

3.2 Implementing a STEM to 'C' precompiler

The STEM to 'C' precompiler, **stetoc**, was designed as a 3-phase translator as illustrated in Figure 3.4. This precompiler takes a STE declaration file as input and outputs compilable 'C' code. Examples of two STE declaration files can be found in Appendix A.1 and Appendix A.2. In this section we provide an overview of the precompiler.

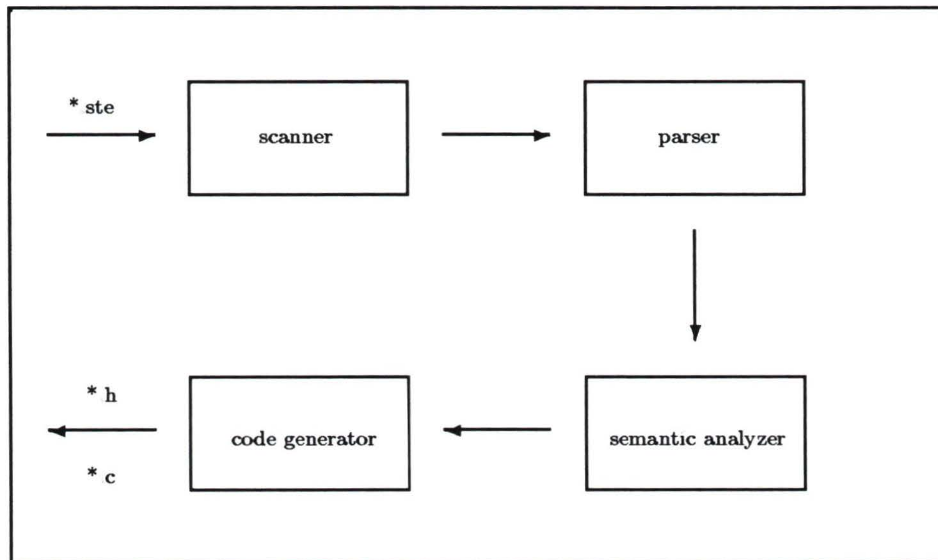


Figure 3.4: Phases of the STEM to 'C' precompiler.

3.2.1 Scanner

The `lex`[3] utility was used to develop a scanner to recognize STE type declarations. The `lex` description file for the scanner (ste.l), is given in Appendix B.1. The scanner recognizes the following keywords: `ste`, `state`, `set`, and, `relation`, as

[3] `lex` is a trademark of AT&T Bell Labs

well as a number of punctuation tokens The STE type declaration for the Radar Set Module can be found in Appendix A 1

3.2.2 Parser

Figure 3 5 below presents an abstract grammar which generates a language for STE type declarations Using a variant of this grammar, the **yacc**[4] utility was used to develop a parser to generate STE type declarations The **yacc** description file for the parser (ste y), is given in Appendix B 2

SteDecl	→ **ste** identifier '{' AttributeList '}' ';'
AttributeList	→ Attribute | Attribute AttributeList
Attribute	→ **state** '{' StateList '}' ';' | **relation** RelationDefn | **set** SetDefn
RelationDefn	→ identifier '{' OrderedPairs '}' ';'
SetDefn	→ identifier '{' IdentifierList '}' ';'
StateList	→ StateLiteral | StateLiteral ',' StateList
OrderedPairs	→ OrderedPair | OrderedPair ',' OrderedPairs
OrderedPair	→ '(' StateLiteral '→' StateLiteral ')'
StateLiteral	→ identifier

Figure 3 5 An abstract grammar for STE types declarations

The parser recognizes three primary constructs

[4] **yacc** is a trademark of AT&T Bell Labs

- *State attribute* Which represents K, the *set of states* in the underlying DFA
- *Relation attributes* Which represent the set of relations a where $a \in \Sigma$ In the STE model the elements of the alphabet (Σ) are the events recognized by the state machine
- *Set attributes* These have no counterparts in the underlying DFA *Sets* are primarily used to partition the state space Faulk [4] refers to these attributes as *modes*

3.2.3 Semantic Analyzer

The semantic analyzer for the precompiler

- Ensures that each STE type declaration has exactly one *state attribute* and at least one *relation attribute* Any STE type declaration meeting these requirements describes either a deterministic or non-deterministic finite automaton *Set attributes* are optional
- Identifies reflexive transitions Reflexive transitions have the form $S \xrightarrow{a} S$ where $S \in K, a \in \Sigma$ When reflexive transitions are encountered by the precompiler, it outputs a warning message to the console No other action is taken The presence of reflexive transitions in the STEM can have undesirable side effects See the discussion in Section 3 4 7 3
- Identifies transitions which lead to a non-deterministic state machine Non-deterministic transitions have the form $S \xrightarrow{a} S_1, S \xrightarrow{a} S_2$ where $S, S_1, S_2 \in K, S_1 \neq S_2, a \in \Sigma$ When this condition is encountered by the precompiler,

it outputs a warning message to the console. The precompiler then removes this transition from the transition table.

All of the warning and error messages generated by the semantic analyzer are documented in Table 3.4.

Error	Type	Comments
SA01: Only one STE type definition per file is allowed	Fatal	
SA02: No STATE declaration found	Fatal	Declaration does not contain a *state attribute*.
SA03: Only one STATE declaration is allowed for each STE type	Fatal	
SA04: No RELATION declarations found	Fatal	Declaration does not contain a *relation attribute*.
Line nn: state - **SA05**: leftmember: no such state	Fatal	The left member in the relation's ordered pair (left→right) is not referenced in the *state attribute*.
Line nn: state - **SA06**: rightmember: no such state	Fatal	The right member in the relation's ordered pair (left→right) is not referenced in the *state attribute*.
SA07: Adding the transition 'Relation(state → state)' will create a self-loop, transition added	Warning	
SA08: Adding the transition 'Relation(state1 → state2)' would create a non-deterministic state machine, transition ignored	Warning	
Line nn: state - **SA09**: no such state	Fatal	The element named in the *set attribute* is not referenced in the *state attribute*.

Table 3.4: Warning and error messages generated by the semantic analyzer.

3.2.4 Code Generator

The code generator is the largest component of the precompiler It uses templates to generate each operation and data structure used by a STEM The generator creates two output files, an *interface module* with the '* h' extension and the *implementation module* with the '* c' extension

The data structures required to implement a STEM are described in Section 3 3 A description of each STEM operation is given in Section 3 4 The interface of each STEM operation is given in Appendix D 1

3.3 Code Generation: Data Structures

The code generator creates five data structures which are necessary to manage a state transition event machine

- STE *type constants* A representation of the *state attribute*
- *Transition matrix* A representation of the *relation attributes*
- *Set Table* Required for state machine control
- *Condition Table* Required for state machine control
- STE *variable declaration* Required for STE variable instantiation

3.3.1 STE Type Constants

For every element of the *state attribute* of a STE type declaration, the precompiler will generate a constant in the implementation module This is illustrated in

Listing 3.1 for the state TRKREL.

STE type constants are used by:

- STE variable constructors to instantiate a STE variable (see Section 3.4.1 on page 45).
- The relation inquiry operations to determine which transitions belong to specific relations (see Section 3.4.5 on page 47).
- The *Wait on Condition* synchronization operations (see Section 3.4.7.4 on page 53).

Listing 3.1 STE type constants for radar set module.

```
1  // File: radar.ste
2  state { TRKREL, RNGREL, STBYREL, ... };
3  ...
4  // File: radar.h
5  typedef struct
6  {
7    int    state;  // The current state of STEM
8    lock_t lock;   // Spinlock used for mutual exclusion
9    FSM    *fsm;   // Pointer to STEM
10 } RADARVAR;
11
12 extern const RADARVAR *TRKREL;
13 extern const RADARVAR *RNGREL;
14 extern const RADARVAR *STBYREL;
15 ...
16 // File: radar.c
17 static const RADARVAR TRKREL_CONSTANT = { 1, 0, NILFSM };
18 const RADARVAR *TRKREL = &TRKREL_CONSTANT;
19 ...
```

3.3.2 Transition Matrix

The precompiler enters all of the *relation attributes* for a STE type into a single transition matrix. Since we store this information as a two-dimensional array only deterministic finite automata can be handled by our representation of a STEM. The transition matrix for the Radar Set Module is shown in Listing 3.2. This matrix has already been presented in another form as Table 3.1.

The primary use of the transition matrix is to make the next state computation in response to state machine events, such as those generated by the state transition operations (see Section 3.4.6 on page 48).

Listing 3.2 Transition matrix for radar set module

```
1  // File: radar.ste
2  relation track { (rngnot→trknot), (rngrel→trkrel), ... };
3  ...
4  // File: radar.c
5  #define NUMSTATES    7
6  #define NUMRELATIONS 6
7
8  // −1 codes for No Transition
9  int Radar[NUMSTATES][NUMRELATIONS] =
10 {
11   -1,  4, -1,  2,  3,  7,
12   -1,  5,  1, -1,  3,  7,
13   -1,  6,  1,  2, -1,  7,
14    1, -1, -1,  5,  6,  7,
15    2, -1,  4, -1,  6,  7,
16    3, -1,  4,  5, -1,  7,
17   -1, -1, -1, -1, -1, -1
18 };
```

3.3.3 Set Table

The precompiler converts all *set attributes* into bitstrings which are represented by the SET data type (see Appendix D.4 on page 151). To assist in state machine control the precompiler also generates a SETTABLE object. The set table serves two purposes:

- It contains a reference to each *set attribute*. These references are used by the set inquiry operations (see Section 3.4.4 on page 47) to determine the set membership of STE variables.
- It is used by the state machine to re-evaluate *set conditions* when a machine event occurs.

Each set table entry has two members:

set: References a *set attribute* in the STE type declaration.

cond: This member holds the result of evaluating the following predicate: **cond** $= S \in T$, where $S \in K$ is the current state of the STEM and T is the associated set in the *set table*.

State transitions force a re-evaluation of each entry in the *set table*. This evaluation is performed by the transition manager update operation (see Section 4.3.4). The set table for the Radar Set Module is shown in Listing 3.3.

Listing 3.3 Set table for radar set module

```
1  typedef struct
2  {
3    SET     *set, // Runtime structure associated with a set
4    Boolean cond, // True  current state is a member of this set
5  } SETTABLE,
6
7  SETTABLE SetTable [ ] =
8  {
9    &RangingSet,      False, // set ranging
10   &IsstandbySet,    False, // set isstandby
11   &IsreliableSet,   False, // set isreliable
12   &NotreliableSet,  False, // set notreliable
13   &TrackingSet,     False  // set tracking
14 },
```

3.3.4 Condition Table

The precompiler uses the information in the transition matrix to generate a *condition table* The condition table is used by the state machine to re-evaluate *relation conditions* when a machine event occurs The condition table is required for the implementation of the wait on condition synchronization operations (see Section 3.4.7.4 on page 53) Each table entry contains four members

relation: Relation attribute index

RHState: A state reachable from the current state of the STEM

handle: Index of the semaphore for this condition

cond: This member holds the result of evaluating the following predicate **cond** $= (S \rightarrow S') \in R$, where $S \in K$ is the current state of the STEM, $S' \in K$ is a state reachable from S (`RHState`) and R is the associated **relation** in the *condition table*

State transitions force the re-evaluation of each entry in the *condition table*. This evaluation is performed by the transition manager update operation (see Section 4.3.4). The condition table for the Radar Set Module is shown in Listing 3.4.

Listing 3.4 Condition table for radar set module

```
1  typedef struct
2  {
3    int     relation, // Relation attribute index
4    int     RHState,  // A state reachable from next state
5    int     handle,   // Semaphore index for this condition
6    Boolean cond,     // True: Relation(nextState->RHState) holds
7  } CONDTABLE;
8
9  CONDTABLE CondTable [ ] =
10 {
11   RELIABLERELATION, 2,  1, False, // relation reliable
12   RELIABLERELATION, 1,  2, False,
13   RELIABLERELATION, 3,  3, False,
14   ...
15   STANDBYRELATION,  6, 11, False, // relation standby
16   STANDBYRELATION,  3, 12, False,
17   FAILURERELATION,  7, 13, False  // relation failure
18 };
```

3.3.5 STE Variables

A STE variable is created using a STE variable constructor (see Section 3.4.1 on page 45). STE variables contain three members:

state:	The current state of the STEM.
lock:	A spinlock to guarantee mutual exclusion.
fsm:	The STEM control structure.

The FSM data structure is described in Section 4.3.1. After instantiation the fsm member contains:

- Dimensioning information for the state machine.
- A set of STE semaphore arrays.
- A copy of the *set table* for this STE type.
- A copy of the *condition table* for this STE type.
- A reference to the *transition matrix* for this STE type.

Listing 3.5 shows the declaration of **RADARVAR**, the STE variable type created by the code generator for the Radar Set Module DFA.

Listing 3.5 Radar STE variable.

```
1  typedef struct
2  {
3    int     state;   // The current state of STEM
4    lock_t  lock;    // Spinlock used for mutual exclusion
5    FSM     *fsm;    // Pointer to STEM
6  } RADARVAR;
```

3.4 Code Generator: STE Operations

The code generator creates the following categories of state machine operations:

- STE *variable constructor*. Constructors are used to instantiate STE variables of a specific type.
- STE *variable destructor*. Destructors are used to destroy STE variables of a specific type.

- STE *variable update operation* The update operation is used whenever a state transition is requested
- *Set inquiry operations* Set inquiry operations are predicate operations which determine if the current state of a STE variable is a member of a specific set
- *Relation inquiry operations* Relation inquiry operations are predicates which determine if a given transition is a member of a specific relation
- *State transition operations* State transition operations are used to request a state change
- *Synchronization operations* Synchronization operations are used by processes when they must wait for specified conditions to hold

A summary of the STE operations generated by the code generator is given in Table 3 5 In this section we describe the implementation of these operations using the Radar Set Module as an example The programming interface of STE operations can be found in Appendix D 1 on page 144

Operation Category	**Faulk and Parnas**[5]	**stetoc**[6] [7]
Variable Constructor		Create*Type*(p)
Variable Destructor		Delete*Type*(p)
Variable Updater		Update*Type*(p,relation)
Set Inquiry Operation	*Set*(p)	Is*Set*(p)
Relation Inquiry Operation	Is*Relation(p,q)*	Is*Relation(p,q)*
State Transition Operation	*Relation(p)*	*Relation(p)*
Synchronization Operation Wait on Transition Event	await@*Relation(p)*	awaitEvt*Relation(p)*
Synchronization Operation Wait on Call	await@=*Relation(p)*	awaitCall*Relation(p)*
Synchronization Operation Wait on Set Transition	await@T *Set(p)* await@F *Set(p)*	awaitSett*Set(p)* awaitSetf*Set(p)*
Synchronization Operation Wait on Condition	await T *Relation(p,q)* await F *Relation(p,q)*	awaitCndt*Relation(p,q)* awaitCndf*Relation(p,q)*
Synchronization Operation Wait on Set Membership	await T *Set(p)* await F *Set(p)*	awaitMbrt*Set(p)* awaitMbrf*Set(p)*

Table 3.5 STE operations

[5] Names of operations in Faulk and Parnas[2, 3]

[6] Names of operations generated by `stetoc`

[7] *Type* = a STE type, *Set* = a STE set attribute, *Relation* = a STE relation attribute p,q = STE variables or constants

3.4.1 STE Variable Constructor

Each STE type requires a variable constructor. If the STE declaration is named *Type*, the code generator will create a constructor template named `CreateType()`. In Listing 3.6 we show the constructor `CreateRadar(p)` for the Radar Set Module. The constructor uses the current state of a reference object, `p`, and a generic constructor (`CreateMachine()`, see Section 4.3 on page 78) to initialize the STE variable. The generic constructor requires dimensioning information (`NUMSETS`, `NUMSTATES`, `NUMRELATIONS`, `NUMCONDITIONS`) as well as references to the STEM's *set table*, *condition table*, and, *transition matrix*.

Listing 3.6 Variable constructor `CreateRadar()`

```
1  RADARVAR *CreateRadar( RADARVAR *p )
2  {
3    int CurState = p→state;
4    RADARVAR *ste;
5
6    if ((ste = (RADARVAR *) malloc(sizeof(RADARVAR))) == NULL) then
7      return( NILRADAR );
8    end if
9
10   ste→state = CurState;
11
12   if ((ste→fsm = CreateMachine( NUMSETS, NUMSTATES, NUMRELATIONS,
13   NUMCONDITIONS, CurState, SetTable, Radar, CondTable )) == NILFSM)
     then
14     free( ste );
15     return( NILRADAR );
16   end if
17
18   return( ste );
19 }
```

3.4.2 STE Variable Destructor

Each STE type requires a variable destructor. If the STE declaration is named *Type*, the code generator will create a destructor template named `DeleteType()`. In Listing 3.7 we show the destructor `DeleteRadar(p)` for the Radar Set Module. The STE destructor uses a generic destructor (`DeleteMachine()`, see Section 4.3 on page 78) to recover the resources allocated to the STE variable.

Listing 3.7 Variable destructor `DeleteRadar()`

```
1  int DeleteRadar( RADARVAR *p )
2  {
3    int restarts = DeleteMachine((STEOBJECT *)p );
4    free( p );
5    return( restarts );
6  }
```

3.4.3 STE Variable Update Operation

Each STE type requires a variable update operation. If the STE declaration is named *Type*, the code generator will create an update template named `UpdateType()`. In Listing 3.8 we show the update operation `UpdateRadar()` for the Radar Set Module. The update operation defers the actual update to a generic update operation (`UpdateMachine()`, see Section 4.3 on page 78). The generic update function requires a reference to an STE variable, `p`, and the identifier for the *event* generating the update.

to wait for two transitions of the STEM. A STE semaphore can only be used to wait for one specified transition event.

3.4.7.1 Wait on Transition Event

The wait on transition synchronization operation is used when a process should wait for a transition in the specified relation. If a *relation attribute* is named *Relation*, the code generator will create the template named `awaitEvtRelation()`. In Listing 3.12 we show the wait on transition synchronization operation `awaitEvtTrack(p)` for the Radar Set Module.

Processes suspended by the synchronization operation associated with relation R will be restarted when the STEM receives any event which causes a transition $(S \rightarrow S') \in R$, where S is the current state of the STEM and S' is the next state of the STEM.

Listing 3.12 Wait on transition event synchronization operation `awaitEvtTrack()`.

```
1  int awaitEvtTrack( RADARVAR *p )
2  {
3    sema_t *sema = &p->fsm->EvtSema[TRACKRELATION];
4    return( dop_pass( sema ));
5  }
```

3.4.7.2 Wait on Call

The wait on call synchronization operation is used when a process should block until the corresponding state transition operation is called (see Section 3.4.6 on page 48). If a *relation attribute* is named *Relation*, the code generator will create the template named `awaitCallRelation()`. In Listing 3.13 we show the wait on

call synchronization operation `awaitCallTrack(p)` for the Radar Set Module. Processes using the `awaitCallTrack(p)` synchronization operation would be blocked until the `Track(p)` state transition operation is invoked by a second process.

Listing 3.13 Wait on call synchronization operation `awaitCallTrack()`

```
1  int awaitCallTrack( RADARVAR *p )
2  {
3    sema_t *sema = &p->fsm->CallSema[TRACKRELATION];
4    return( dop_pass( sema ));
5  }
```

3.4.7.3 Wait on Set Transition

The wait on set transition synchronization operations block processes unconditionally in the following situations:

- until the state of a STE variable changes to an element *inside* a set.
- until the state of a STE variable changes to an element *outside* a set.

If a *set attribute* is named *Set*, the code generator will create two templates named `awaitSettSet()` and `awaitSetfSet()`. In Listing 3.14 we show the wait on set transition operations `awaitSettTrack(p)` and `awaitSetfTrack(p)` for the Radar Set Module.

The `awaitSettSet(p)` form of the operation forces a process to wait until the value of an STE variable changes from a value *outside* the set to a value *inside* the set.

The `awaitSetfSet(p)` form of the operation forces a process to wait until the value of an STE variable changes from a value *inside* the set to a value *outside* the set.

Faulk[4] expresses some difficulty in working with these operations. This difficulty arises when the STEM contains reflexive transitions ($S \xrightarrow{\text{event}} S$). Under these circumstances his implementation of the set transition synchronization operations appears unable to distinguish between the one transition case (*inside* → *inside*) and the two transition case (*inside* → *outside* → *inside*).

Our implementation of the set transition synchronization operation (see Listing 3.14), requires two semaphores to implement the two transition case. Our transition manager has no problem handling reflexive transitions (see Section 4.3).

Listing 3.14 Wait on set transition synchronization operation `awaitSettTrack()`

```
1  int awaitSettTrack( RADARVAR *p )
2  {
3    sema_t *ssema = &p→fsm→SSettSema[TRACKINGSET];
4    sema_t *lsema = &p→fsm→LSettSema[TRACKINGSET];
5
6    if (IsTracking( p ) == True ) then
7      return( dop_pass( lsema ));
8    else
9      return( dop_pass( ssema ));
10   end if
11 }
12
13 int awaitSetfTrack( RADARVAR *p )
14 {
15   sema_t *ssema = &p→fsm→SSetfSema[TRACKINGSET];
16   sema_t *lsema = &p→fsm→LSetfSema[TRACKINGSET];
17
18   if (IsTracking( p ) == True ) then
19     return( dop_pass( ssema ));
20   else
21     return( dop_pass( lsema ));
22   end if
23 }
```

3.4.7.4 Wait on Condition

The wait on condition synchronization operations are used to choose between blocking and not blocking an executing process depending on the value of a condition in the *condition table*. If a *relation attribute* is named *Relation*, the code generator will create two templates named `awaitCndtRelation()` and `awaitCndfRelation()`. In Listing 3.15 we show the wait on condition synchronization operations `awaitCntt-Track(p,q)` and `awaitCntfTrack(p,q)` for the Radar Set Module.

Each *relation attribute* contributes elements to the *condition table* subject to the following constraint:

$$(a \rightarrow b) \in R \text{ is a condition iff } (c \rightarrow b) \notin R \text{ for any } c \neq a$$

where $a, b, c \in K$, the state set of the STEM and R is a relation attribute. This indicates that not all combinations of input parameters `p` and `q` are valid. As an aid to implementing these synchronization operations, the precompiler generates a matrix, `CondMatrix[numrelations][numstates]`, which contains the legal combinations.

The `awaitCndtRelation(p,q)` form of the operation allows a process to wait if the relation *does not* hold between STE variables `p` and `q`. Processes suspended by this synchronization operation will be restarted when the relation $(S \rightarrow S') \in R$ holds for S the current state of STE variable `p` and S' the current state of STE variable `q`.

The `awaitCndfRelation(p,q)` form of the operation allows a process to wait if the relation *does* hold between STE variables `p` and `q`. Processes suspended by this synchronization operation will be restarted when the relation $(S \rightarrow S') \in R$

does not hold for S the current state of STE variable p and S' the current state of STE variable q.

Listing 3.15 Wait on condition synchronization operation `awaitCndtTrack()` and `awaitCndfTrack()`

```
1  int awaitCndtTrack( RADARVAR *p1, RADARVAR *p2 )
2  {
3    sema_t *sema;
4    int handle;
5
6    if ( (handle = CondMatrix[TRACKRELATION][(p2→state)−1]) <= 0 )
     then
7      return( -1 );
8    end if
9    sema = &p1→fsm→CndtSema[handle−1];
10   return( dop_pass( sema ));
11 }
12
13 int awaitCndfTrack( RADARVAR *p1, RADARVAR *p2 )
14 {
15   sema_t *sema;
16   int handle;
17
18   if ( (handle = CondMatrix[TRACKRELATION][(p2→state)−1]) <= 0 )
     then
19     return( -1 );
20   end if
21   sema = &p1→fsm→CndfSema[handle−1];
22   return( dop_pass( sema ));
23 }
```

3.4.7.5 Wait on Set Membership

The wait on set membership synchronization operations are used to choose between blocking and not blocking an executing process depending on the value of a condition in the *set table*. If a *set attribute* is named *Set*, the code generator will cre-

ate two templates named `awaitMbrtSet()` and `awaitMbrfSet()` In Listing 3.16 we show the wait on condition synchronization operations `awaitMbrtTracking(p)` and `awaitMbrfTracking(p)` for the Radar Set Module

The `awaitMbrtSet(p)` form of the operation allows a process to wait if the current state of the STE variable p *is not* a member of *Set* Processes suspended by this synchronization operation will be restarted when the current state of the STE variable p changes to a value inside of *Set*

The `awaitMbrfSet(p)` form of the operation allows a process to wait if the current state of the STE variable p *is* a member of *Set* Processes suspended by this synchronization operation will be restarted when the current state of the STE variable p changes to a value outside of *Set*

Listing 3.16 Wait on set membership synchronization operation `awaitMbrtTracking()` and `awaitMbrfTracking`

```
1  int awaitMbrtTracking( RADARVAR *p )
2  {
3    sema_t *sema = &p→fsm→MbrtSema[TRACKINGSET],
4    return( dop_pass( sema )),
5  }
6
7  int awaitMbrfTracking( RADARVAR *p )
8  {
9    sema_t *sema = &p→fsm→MbrfSema[TRACKINGSET],
10   return( dop_pass( sema )),
11 }
```

3.5 STE Variables and the STE Model

In Sections 3.3 and 3.4 we have described how the **stetoc** precompiler generates a STEM implementation from a STE declaration. Figure 3.6 illustrates a STE variable in terms of its data structures and STE operations. It is clear that this implementation preserves all of the passive components in the STE model presented in Section 1.1. The deterministic finite automaton ($\mathcal{M}$) in the STE model is represented by the *state variable* and *transition matrix* in the STE variable. The condition table in the STE model is represented by the *set conditions* (α, β) and *relation conditions* (ψ, ω) in the STE variable. Each condition has an associated semaphore.

Figure 3.6 also suggests how threads might interact with STE variables. Threads which are performing the event-detecting role in the STE model, interact using the *state transition operations* (see Section 3.4.6). Threads which are performing the condition-enabled role in the STE model, block on condition semaphores by using the *synchronization operations* (see Section 3.4.7).

3.6 Summary

This chapter described the output of a precompiler which can convert the description of a state transition event machine into compilable 'C' code. The abstract data type generated by the precompiler has provision for semaphores used by the synchronization operations, a set table required by the transition manager and a condition table required by the transition manager. The precompiler provides all state inquiry, state transition and synchronization operations described by Faulk

and Parnas [3]

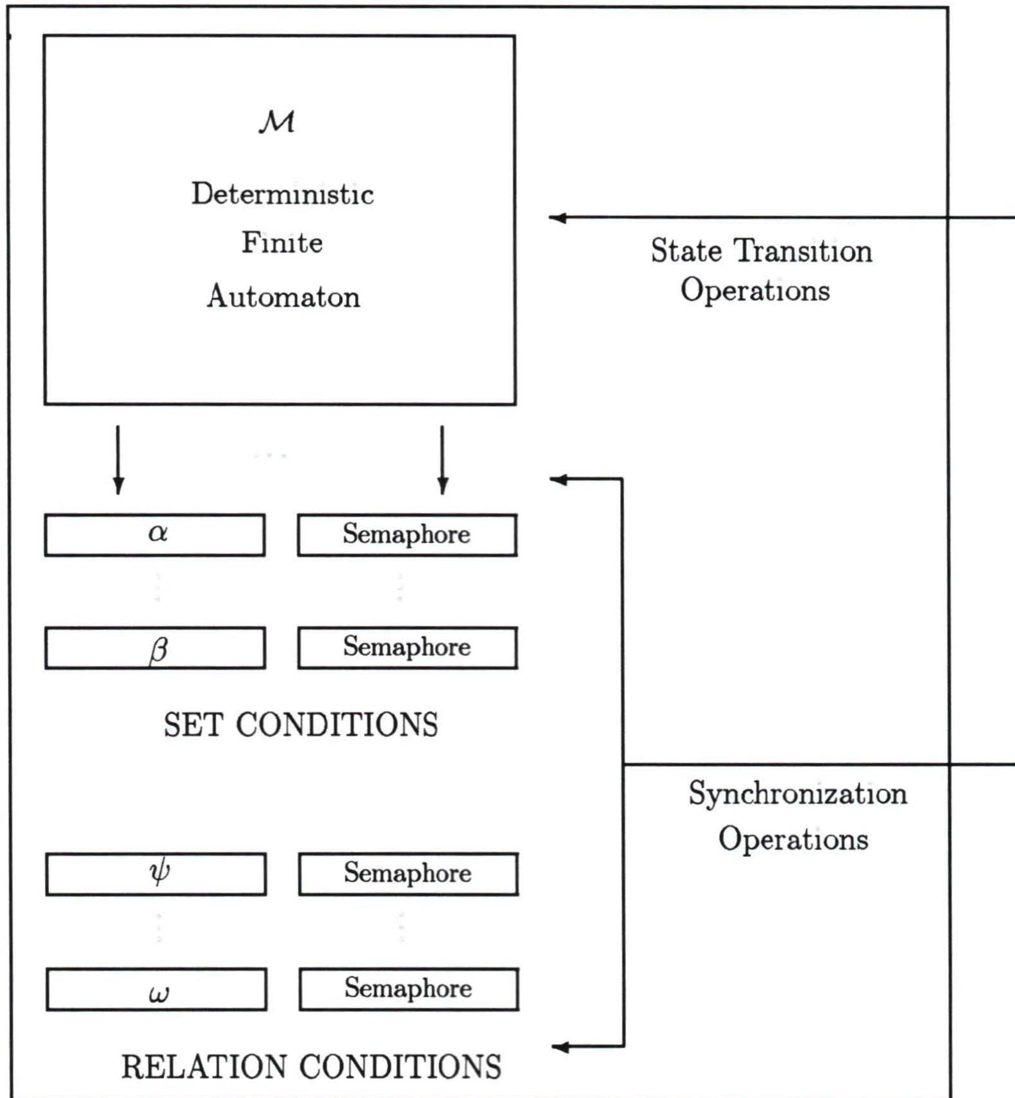


Figure 3.6: An implementation of a STE variable.

Chapter 4

Implementing Low-Level Synchronization Operations

4.1 VSTa: An Experimental Microkernel

The run-time services required by our low-level synchronization operations (STE d-operations) are provided by the VSTa microkernel [1] VSTa is an experimental kernel which attempts to blend the design of a microkernel with the system architecture of Plan 9 [18] [2] The result is a small privileged kernel running user-mode processes to provide system services (device drivers, filesystems, etc)

In the following list we have summarized some of VSTa's more important features:

Support for Threads: VSTa supports a simple threads model Its interface al-

[1] Valencia Simple Tasker Presently we are working with VSTa 1 5 2, released on December 29, 1996

[2] Plan 9 is a trademark of AT&T Bell Labs

lows for thread creation, cancellation, and, barrier synchronization. Thread join and detachment are not supported at the current revision level. VSTa supports a `syscall()` interface to provide access to kernel services. As threads execute, they switch freely from user-mode to kernel-mode to execute these services.

Messaging: VSTa structures the exchange of messages between clients and servers. Clients request connection to a specific server and are granted a unique `port` for all subsequent communications with that server. VSTa supports scatter/gather lists for messages. Each element in the scatter/gather list of the message is visible in the server's address space when the server receives the message. All message data is visible in the client's address space before the client request is complete.

Real-time Facilities: VSTa is not a real-time operating system. But the process of *microkernelization*, moving functionality from the kernel to user-mode based servers, often requires features associated with real-time systems:

- process memory locking is necessary in order to allow device drivers to run as user processes.
- low latency process dispatch is necessary to allow interrupt service code to run in deterministic amounts of time.
- non-degrading scheduling priorities which permit critical system services to respond to multiple clients without being penalized for their apparent heavy CPU usage.

Scheduling: VSTa supports a priority driven scheduler. The scheduler services three queues, from lowest to highest priority: **BG** (background queue), **TIMESHARE** (timeshare queue), and, **RT** (real-time queue). Within each queue threads receive a fixed timeslice. Preemption can occur between queues. For example, a thread placed on the **RT** queue would immediately preempt a thread running in either the **TIMESHARE** or **BG** queues.

Filesystem Interface: VSTa servers provide their services through a filesystem-like interface. The format of the messages sent through the microkernel is standard and implements a filesystem protocol similar to Plan 9's 9P protocol.

4.2 STE D-Operations

At least two approaches were considered in the design of our low level synchronization operations (d-operations).

Design a VSTa server: VSTa servers execute as user-mode processes. This suggests that requests for semaphore services would be converted to IPC requests using the native messaging system. We expect that any message based protocol would lead to high-latency semaphore operations.

Extend the microkernel: In VSTa, access to kernel functions is through the **syscall()** interface. This allows processes to enter kernel-mode to execute kernel operations on their own behalf. We believe that this approach will result in low-latency semaphore operations.

To create the semaphore services required by our high-level synchronization operations, we extended the VSTa `syscall()` interface to include the operations given in Table 4.1. Our operations, which we refer to as d-operations or STE d-operations, represent a specialization of VSTa semaphores. The data structure accessed by the d-operations is referred to as a STE semaphore.

A description of the programming interface for the d-operations can be found in Appendix D.2. In Sections 4.2.1–4.2.5 we describe how each d-operation works.

Operation	**Description**
dop_create	Create a semaphore.
dop_destroy	Destroy a semaphore.
dop_pass	Semaphore passage operation.
dop_up	Semaphore opening operation.
dop_down	Semaphore closing operation.
dop_updown	Atomic open-close operation.
dop_movesema	Semaphore morphing operation.

Table 4.1: STE d-operations.

Table 4.2 contains a summary of the VSTa kernel functions used in the implementation of the d-operations. These functions fall into the following categories:

Semaphores and spinlocks. This is VSTa's semaphore interface. These functions are only available in the kernel. STE d-operations are available to all user-mode processes.

Hash functions: This is VSTa's hash table implementation. These functions are available to the kernel and all user-mode processes (`libsrv.a`).

Scheduler functions: We use two entry points into VSTa's scheduler. `swtch()` is used to perform context switches, and, `lsetrun()` returns a sleeping thread to a scheduling queue.

Error handling: `err()` is used to register a run-time error against an errant thread. A user-mode process can retrieve this information by using the `strerror()` function in the standard library (`libc.a`).

Operation	Visibility[3]	Description
Semaphores and Spinlocks		
p_lock	K	Acquire a spinlock
v_lock	K	Release a spinlock
p_sema	K	Acquire a semaphore
v_sema	K	Release a semaphore
q_sema	K	Queue a thread on a semaphore
dq_sema	K	Remove a thread from a semaphore
init_sema	K	Initialize a semaphore
Hash Functions		
hash_alloc	U	Create a hash data structure
hash_insert	U	Insert (key/value) pair into hash table
hash_delete	U	Delete a node from hash table
hash_lookup	U	Lookup a node based on its key
Scheduler Functions		
swtch	K	Switch to another thread
lsetrun	K	Move thread to run queue
Error Handling		
err	K	Register a run-time error

Table 4 2 VSTa functions used in the design of STE d-operations

[3]Visibility, K=visible only in the kernel, U=user library

4.2.1 Data Structures

Listing 4.1 summarizes the data structures and type definitions necessary to use VSTa spinlocks and semaphores. VSTa spinlocks may be acquired either with interrupts disabled (SPLHI) or enabled (SPLO). When a VSTa semaphore is acquired the caller must decide if the process will be allowed to respond to system events while sleeping (PRICATCH) or to sleep without interruption (PRIHI).

To distinguish STE semaphores, our interface supports a handle's strategy (types `sid_t` or `SEMAPHORE`). Before d-operations can be performed on STE semaphores, a semaphore handle must be acquired using the `dop_create()` operation (see Section 4.2.3).

Listing 4.1 Data structures: VSTa spinlocks and semaphores

```
1  // VSTA Spinlock
2  typedef struct lock
3  {
4    uchar   l_lock;
5  } lock_t;
6
7  typedef unsigned int spl_t;
8  #define SPLO // Spin with interrupts enabled
9  #define SPHI // Spin with interrupts disabled
10
11  spl_t p_lock( lock_t *, spl_t ); // Acquire a spinlock
12  void v_lock( lock_t *, spl_t ); // Release a spinlock
13
14  // VSTA Semaphore
15  typedef struct sema
16  {
17    lock_t  s_lock;    // Mutex for semaphore
18    int     s_count;   // Count
19    thread *s_sleepq;  // List of sleeping threads
20  } sema_t;
21
22  typedef unsigned int pri_t;
23  #define PRIHI // Sleep uninterruptibly
24  #define PRICATCH // Sleep interruptibly
25
26  int p_sema( sema_t *, pri_t ); // Acquire a semaphore
27  void v_sema( sema_t * ); // Release a semaphore
28
29  // STE Semaphore
30  typedef long sid_t; // Handle to STE semaphore
31  typedef sid_t SEMAPHORE; // Alias for STE semaphore
32  #define S_OPENED // Initialize semaphore in non-blocked state
33  #define S_CLOSED // Initialize semaphore in blocked state
```

4.2.2 Subsystem Initialization

The STE semaphore subsystem is initialized by the kernel at system startup by executing function init_semaphore() which is shown in Listing 4 2 The structures initialized here include a hash table (sid_hash), and a VSTa semaphore (sid_sema) which acts as a mutex protecting the hash table

Listing 4 2 STE semaphores subsystem initialization

```
1  sema_t  sid_sema = 0,   // Mutex for hash table
2  hash    *sid_hash = 0, // Hash table for semaphores
3
4  void init_semaphore( void )
5  {
6    init_sema( &sid_sema ),
7    sid_hash = hash_alloc( 64 ),
8  }
```

4.2.3 dop_create and dop_destroy

To acquire a STE semaphore a user must call the dop_create() interface function This function takes a single parameter, cnt, which determines the initialization state of the semaphore If the initial value is S_OPENED the semaphore will be initialized in a non-blocking state If the initial value is S_CLOSED the semaphore will be initialized in a blocking state

In Listing 4 3 we show our implementation of the create STE operation, dop_-create() dop_create() executes in the following manner

- A VSTa semaphore is allocated then initialized (see dop_init() in Listing 4 5) with the cnt parameter If allocation fails, register the ENOMEM run-time error with the kernel and abort the operation

- Acquire the `sid_sema` mutex. This is required to insert the STE semaphore into the hash table.

- Generate a unique STE semaphore identifier (see `alloc_sid()` in Listing 4.5).

- Insert the VSTa semaphore in the hash table (`sid_hash`) using the STE semaphore identifier as key. If this operation fails, release allocated resources, register the ENOMEM run-time error with the kernel then abort the operation.

- Release the `sid_sema` mutex.

- Return the STE semaphore identifier to caller.

Listing 4.3 Create STE semaphore operation dop_create()

```
1  sid_t dop_create( int cnt )
2  {
3    struct sema *s;
4    sid_t nsid;
5
6    if ((s = malloc( sizeof(sema_t))) == NULL) then
7      return( err(ENOMEM) );
8    end if
9
10   dop_init( s, cnt );
11   p_sema( &sid_sema, PRIHI );
12   nsid = alloc_sid();
13
14   if ( hash_insert( sid_hash, nsid, s ) == 1 ) then
15     free( s );
16     v_sema( &sid_sema );
17     return( err(ENOMEM) );
18   end if
19
20   v_sema( &sid_sema );
21   return( nsid );
22 }
```

When an application has no further use of a STE semaphore, it must be returned to the kernel using the dop_destroy() operation (see Listing 4.4). This operation's single parameter must identify a STE semaphore previously acquired with the dop_create() operation. dop_destroy() executes in the following manner:

- Acquire the sid_sema mutex. This is required anytime we access the semaphore hash table.
- Validate the input parameter, sid, by using hash_lookup() to find the sema-

phore identifier in semaphore hash table If this operation fails, register the ESRCH run-time error with the kernel then abort the operation

- If the STE semaphore identifier is valid, but the semaphore queue is not empty (threads still waiting), register the EEXIST run-time error with the kernel then abort the operation

- If all checks pass, remove the semaphore from the hash table

- Release the sid_sema mutex Free the space allocated to the VSTa semaphore

Listing 4.4 Destroy STE semaphore operation dop_destroy()

```
1  int dop_destroy( sid_t sid )
2  {
3    struct sema *s,
4
5    p_sema( &sid_sema, PRIHI ),
6
7    if ((s = hash_lookup( sid_hash, sid )) == NULL) then
8      v_sema( &sid_sema ),
9      return(err(ESRCH) ),
10   end if
11
12   if ( s→s_sleepq ) then
13     v_sema( &sid_sema ),
14     return( err(EEXIST) ),
15   end if
16
17   hash_delete( sid_hash, sid ),
18   v_sema( &sid_sema ),
19   free( s ),
20   return( 0 ),
21 }
```

Listing 4.5 Create STE semaphore utility functions

```
1  void dop_init( struct sema *s, int cnt )
2  {
3    s->s_count = cnt;
4    s->s_sleepq = 0;
5    init_lock( &s->s_lock );
6  }
7
8  sid_t alloc_sid( void )
9  {
10   static ulong rotor = 0L;
11
12   rotor += 1;
13   while ( hash_lookup( sid_hash, rotor )) do
14     rotor += 1;
15     if  ( rotor >= 65535L )  then
16       rotor = 1L;
17     end if
18   end while
19   return((sid_t) rotor );
20 }
```

4.2.4 dop_pass, dop_up and dop_down

In this section we discuss our implementation of the d-operations introduced by Belpaire and Wilmotte [5]. Our implementation contains a fourth primitive operation, `dop_updown()`. This operation is used to atomically combine the opening and closing operations and is denoted as `[dop_up() ; dop_down()]`. This operation is required to implement the conditional synchronization operations (see Section 3.4.7).

Listing 4.6 shows our implementation of the semaphore passage operation, `dop_pass()`. Our implementation is modeled after the VSTa `p_sema()` semaphore

primitive dop_pass() executes in the following manner

- Acquire the sid_sema mutex to gain access to the semaphore hash table
- Validate the input parameter, sid, by looking up the STE semaphore in the semaphore hash table If this operation fails, register the ESRCH run-time error with the kernel then abort the operation
- Acquire the semaphore lock (p_lock) At the same time release the sid_sema mutex This allows concurrent access to the hash table while guarding access to a specific STE semaphore
- If the semaphore value is nonnegative, we release the semaphore lock and return with a return code of 0
- Prepare the executing thread for sleeping The thread is suspended interruptibly (PRICATCH) This allows the thread to respond to system events
- Acquire the scheduler lock Release the semaphore lock Enter the scheduler to perform a context switch Current thread does not return until the semaphore is signaled
- When restarted, thread can determine if passage was successful If the thread was signaled by the operating system, it leaves the passage operation with a return code of 1 If the thread was signaled by from the dop_up() operation, it leaves the passage operation with a return code of 2

Listing 4.6 Semaphore passage operation dop_pass()

```
1: int dop_pass( sid_t sid )
2: {
3:   pri_t p = PRICATCH;
4:   struct thread *t;
5:   sema_t *s;
6:
7:   p_sema( &sid_sema, PRIHI );
8:
9:   if ((s = hash_lookup( sid_hash, sid )) == NULL) then
10:    v_sema( &sid_sema );
11:    return( err(ESRCH) );
12:  end if
13:
14:  p_lock( &s→s_lock, SPLHI );
15:  v_sema( &sid_sema );
16:
17:  if (s→s_count >= 0) then
18:    v_lock( &s→s_lock, SPL0);
19:    return( 0 );
20:  end if
21:
22:  // Prepare thread for sleep.
23:  t = curthread; t→t_wchan = s;
24:  t→t_nointr = (p == PRIHI); t→t_intr = 0;
25:  q_sema(s, t);
26:  p_lock( &runq_lock, SPLHI_SAME);
27:  v_lock( &s→s_lock, SPLHI_SAME);
28:  t→t_state = TS_SLEEP;
29:  swtch();
30:
31:  // We're back. Return 1=system event, 2=received signal.
32:  t→t_nointr = 0;
33:
34:  if (t→t_intr) then
35:    return( 1 );
36:  end if
37:
38:  return( 2 );
39: }
```

Listing 4.7 shows our implementation of the semaphore opening operation, dop_up(). Our implementation is modeled after the VSTa v_sema() semaphore primitive. dop_up() executes in the following manner.

- Acquire the sid_sema mutex to obtain access to the semaphore hash table.
- Validate the input parameter, sid, by looking up the STE semaphore in the semaphore hash table. If this operation fails, register the ESRCH run-time error with the kernel then abort the operation.
- Acquire the semaphore lock (p_lock). At the same time release the sid_sema mutex. This allows concurrent access to the hash table while guarding access to a specific STE semaphore.
- Increment the value of the semaphore.
- If the value of the semaphore is nonnegative, restart all threads waiting on the semaphore by using lsetrun() to return the threads to their scheduling queue. Track the number of threads restarted.
- Release the semaphore lock and return the thread restart count to the caller.

Listing 4.7 Semaphore opening operation dop_up()

```
1  int dop_up( sid_t sid )
2  {
3    struct thread *t;
4    spl_t spl;
5    int waiters = 0;
6    sema_t *s;
7
8    p_sema( &sid_sema, PRIHI );
9
10   if ((s = hash_lookup( sid_hash, sid )) == NULL) then
11     v_sema( &sid_sema );
12     return( err(ESRCH) );
13   end if
14
15   spl = p_lock( &s→s_lock, SPLHI );
16   v_sema( &sid_sema );
17   s→s_count += 1;
18
19   if ( s→s_count >= 0 ) then
20     while ( s→s_sleepq ) do
21       waiters++; // Keep track of the number of threads restarted
22       t = s→s_sleepq;
23       dq_sema(s, t);
24       t→t_wchan = 0;
25       p_lock( &runq_lock, SPLHI );
26       lsetrun(t);
27       v_lock( &runq_lock, SPLHI );
28     end while
29   end if
30
31   v_lock( &s→s_lock, spl );
32   return( waiters );
33 }
```

Listing 4.8 shows our implementation of the semaphore closing operation, dop_-down(). dop_down() executes in the following manner:

- Acquire the sid_sema mutex to obtain access to the semaphore hash table.
- Validate the input parameter, sid, by looking up the STE semaphore in the semaphore hash table. If this operation fails, register the ESRCH run-time error with the kernel then abort the operation.
- Acquire the semaphore lock (p_lock). At the same time release the sid_sema mutex. This allows concurrent access to the hash table while guarding access to a specific STE semaphore.
- Decrement the value of the semaphore.
- Release the semaphore lock and return with a return code of 0.

Listing 4.8 Semaphore closing operation dop_down()

```
1  int dop_down( sid_t sid )
2  {
3    spl_t spl;
4    struct sema *s;
5
6    p_sema( &sid_sema, PRIHI );
7
8    if ((s = hash_lookup( sid_hash, sid )) == NULL) then
9      v_sema( &sid_sema );
10     return( err(ESRCH) );
11   end if
12
13   spl = p_lock( &s→s_lock, SPLHI );
14   v_sema( &sid_sema );
15
16   s→s_count -= 1;
17
18   v_lock( &s→s_lock, spl );
19   return( 0 );
20 }
```

4.2.5 dop_movesema

`dop_movesema()`, the semaphore morphing operation, is required in the implementation of the *wait on set transition* synchronization operation (see Section 3.4.7.3 on page 51). Each such operation requires two STE semaphores for correct implementation. We refer to these as the *short semaphore* and the *long semaphore.* When the synchronization operation is executed one of two things can happen. If the operation's condition holds, then the executing thread passes onto the *long semaphore.* Such threads are required to wait for two events; a `@F(condition)` followed by a `@T(condition)`. However, if the operation's condition does not hold initially, then the executing thread passes onto the *short semaphore.* Such threads are required to wait for a single event; `@T(condition)`. When the `@F(condition)` event occurs, threads waiting on the *long semaphore* must be transferred atomically to the associated *short semaphore.* This transfer of threads from one semaphore's queue to another, without any intervening context switches, is referred to as *morphing* [10].

Listing 4.9 shows our implementation of the semaphore morphing operation, `dop_movesema()`. `dop_movesema()` executes in the following manner:

- Acquire the `sid_sema` mutex to obtain access to the semaphore hash table.
- Validate the two input parameters, `src` the identifier of the source semaphore, and, `dst` the identifier of the destination semaphore, by looking up the identifiers in the semaphore hash table. If either operation fails, register the ESRCH run-time error with the kernel then abort the operation.
- Notice that in order to keep this operation atomic we elect to hold the

hash table mutex while moving the source queue The alternative would be to attempt to acquire the semaphore locks on the source and destination semaphores This later approach could lead to a deadlock

- If there are any threads suspended on the source semaphore queue, dequeue the threads and place on the destination semaphore queue Track the number of threads moved

- Release the hash table mutex and return the thread transfer count to the caller

Listing 4.9 Semaphore morphing operation dop_movesema()

```
1  int dop_movesema( sid_t src, sid_t dst )
2  {
3    struct thread *t;
4    sema_t *s, *d;
5    int movers = 0;
6
7    p_sema( &sid_sema, PRIHI );
8
9    if ((s = hash_lookup( sid_hash, src )) == NULL) then
10     v_sema( &sid_sema );
11     return( err(ESRCH) );
12   end if
13
14   if ((d = hash_lookup( sid_hash, dst )) == NULL) then
15     v_sema( &sid_sema );
16     return( err(ESRCH) );
17   end if
18
19   while ( s->s_sleepq ) do
20     movers++;
21     t = s->s_sleepq;
22     dq_sema(s, t);
23     t->t_wchan = d;
24     q_sema( d, t );
25   end while
26
27   v_sema( &sid_sema );
28   return( movers );
29 }
```

4.3 Transition Manager

The transition manager is an interface used to create, destroy, and, update the STEM associated with a STE variable. As discussed in Section 3.4, our precom-

piler, **`stetoc`**, generates three template operations for each STE declaration it processes. `Create`*`Type`*`()`, the variable constructor, `Delete`*`Type`*`()`, the variable destructor, and, `Update`*`Type`*`()` the variable updater. These three operations are written in terms of transition manager operations, namely, `CreateMachine()`, `DeleteMachine()`, and, `UpdateMachine()`. The transition manager operations are summarized in Table 4.3.

Operation	**Description**
CreateMachine	Create STEM
DeleteMachine	Destroy STEM
UpdateMachine	Update STEM
Lock	Acquire a user-mode spinlock
UnLock	Release a user-mode spinlock

Table 4.3: Transition manager operations.

At this point there are two things which need considering:

- In any application, every STE variable uses the same transition manager interface. This requires the operations to be thread safe. To guarantee thread safety, each STE variable contains a user-mode spinlock.

- The transition manager interface is *rarely* required by the application programmer. This is also true for the d-operation interface described in Section 4.2. Application programs use the high-level interface provided by the STE operations (see Sections 3.4.1–3.4.7). This arm's length approach allows

programmers to design applications with complex synchronization requirements using only the high-level synchronization operators.

A description of the programming interface for the transition manager can be found in Appendix D.3. In the remainder of this section we describe the work performed by the transition manager.

4.3.1 Transition Manager Data Structures

Listing 4.10 contains the definitions of the data types required by the transition manager. All STE variables are derived from the **STEOBJECT** data type. A **STEOBJECT** object contains the following three members:

state:	The current state of the STEM.
lock:	A spinlock to guarantee mutual exclusion.
fsm:	The STEM.

Our implementation of the state transition event machine (**fsm** data type), contains the following objects:

- Ten arrays of STE semaphores. After initialization, these arrays will hold the handles to all semaphores used by the high-level synchronization operations.
- Dimensioning information for the STEM. The number of sets, states, relations and conditions found in the STE declaration.
- A copy of the set table for the declared STE type.
- A copy of the condition table for the declared STE type.
- A reference to the transition matrix for the declared STE type.

Listing 4.10 Data structures: state transition event machine.

```
1:  // User-mode spinlock.
2:  typedef unsigned int lock_t;
3:
4:  // STE variable.
5:  typedef struct
6:  {
7:    int    state;   // The current state of the STEM
8:    lock_t lock;    // Spinlock used for mutual exclusion
9:    FSM    *fsm;    // Pointer to the STEM
10: } STEOBJECT;
11:
12: // STE state machine.
13: typedef struct
14: {
15:   SEMAPHORE *EvtSema;   // Semaphore: Wait on Transition Event
16:   SEMAPHORE *CndtSema;  // Semaphore: Wait on Condition (T)
17:   SEMAPHORE *CndfSema;  // Semaphore: Wait on Condition (F)
18:   SEMAPHORE *CallSema;  // Semaphore: Wait on Call
19:   SEMAPHORE *MbrtSema;  // Semaphore: Wait on Set Membership (T)
20:   SEMAPHORE *MbrfSema;  // Semaphore: Wait on Set Membership (F)
21:   SEMAPHORE *SSettSema; // Semaphore: Wait on Set Transit (ST)
22:   SEMAPHORE *LSettSema; // Semaphore: Wait on Set Transit (LT)
23:   SEMAPHORE *SSetfSema; // Semaphore: Wait on Set Transit (SF)
24:   SEMAPHORE *LSetfSema; // Semaphore: Wait on Set Transit (LF)
25:
26:   int       NumSets;       // Number of sets
27:   int       NumStates;     // Number of states
28:   int       NumRelations;  // Number of relations
29:   int       NumConditions;// Number of conditions
30:
31:   SETTABLE  *SetTable;   // Set table associated with STEM
32:   int       **Transition;// Trans. matrix associated with STEM
33:   CONDTABLE *CondTable;  // Condition table associated with STEM
34: } FSM;
```

4.3.2 Creating State Transition Event Machines

`CreateMachine()` is a generic constructor used to create and initialize the STEM component of a STE variable. The major steps in creation and initialization are summarized in Listing 4.11. The detailed initialization of the STE semaphores created for the high-level synchronization operations is given in Table 4.4. As an example, when `InitializeMbrSemaphores()` is executed each semaphore is initialized according to the current state of the *set table*. If the current condition for some *set attribute* is `true`, then the associated semaphore will be initialized with the statement:

```
tsema = dop_create( S_OPENED )
```

Threads *passing* on a semaphore in this state will not be blocked. If the current condition for the same *set attribute* is `false`, then the associated semaphore will be initialized with the statement:

```
tsema = dop_create( S_CLOSED )
```

Threads *passing* on a semaphore in this state will be blocked.

Listing 4.11 Create state transition event machine

```
1  FSM *CreateMachine( int numsets, int numstates, int numrelations, int
   numconditions, int curState, SETTABLE *setTable, int **trans,
   CONDTABLE *condTable )
2  {
3    FSM *fsm = (FSM *) malloc(sizeof(FSM));
4    fsm->Transition = trans;
5
6    SaveDimensionInformation( fsm, numsets, numstates, numrelations,
     numconditions );
7    CreateSetTable( fsm, setTable );
8    CreateConditionTable( fsm, condTable );
9
10   InitializeEvtSemaphores( fsm );
11   InitializeCallSemaphores( fsm );
12
13   InitializeCndSemaphores( fsm, curState );
14   InitializeConditionTable( fsm );
15
16   InitializeMbrSemaphores( fsm, curState );
17   InitializeSetSemaphores( fsm, curState );
18   InitializeSetTable( fsm );
19
20   return( fsm );
21 }
```

Synchronization Category	Value[4]	Initialization
Unconditional Synchronization Operations		
Wait on Transition Event `awaitEvtRelation(p)`		`sema = dop_create(S_CLOSED)`
Wait on Call `awaitCallRelation(p)`		`sema = dop_create(S_CLOSED)`
Wait on Set Transition `awaitSettSet(p)`		`stsema = dop_create(S_CLOSED)` `ltsema = dop_create(S_CLOSED)`
Wait on Set Transition `awaitSetfSet(p)`		`sfsema = dop_create(S_CLOSED)` `lfsema = dop_create(S_CLOSED)`
Conditional Synchronization Operations		
Wait on Condition `awaitCndtSet(p)`	T F	`tsema = dop_create(S_OPENED)` `tsema = dop_create(S_CLOSED)`
Wait on Condition `awaitCndfSet(p)`	T F	`fsema = dop_create(S_CLOSED)` `fsema = dop_create(S_OPENED)`
Wait on Set Membership `awaitMbrtSet(p)`	T F	`tsema = dop_create(S_OPENED)` `tsema = dop_create(S_CLOSED)`
Wait on Set Membership `awaitMbrfSet(p)`	T F	`fsema = dop_create(S_CLOSED)` `fsema = dop_create(S_OPENED)`

Table 4.4 Transition manager: semaphore initialization

4.3.3 Destroying State Transition Event Machines

`DeleteMachine()` is used to destroy the STEM component of a STE variable, and recover all of the resources associated with the state machine. This is accomplished

[4]Value=value of corresponding entry in condition table or set table

This will cause the semaphore to become *closed*, so any thread *passing* on the semaphore will be blocked.

Since our implementation of `UpdateMachine()` uses a spinlock to grant exclusive access to the `STE` variable, all semaphore and condition table updates take place in a single cycle. The threads restarted by `UpdateMachine()` cannot begin execution until the spinlock is released. This implementation is subject to the *intercepted wakeup problem* [10]. If `UpdateMachine()` restarts a large number of threads in one cycle, it is possible for an event-detecting thread to signal an event which alters the condition table before restarted threads are scheduled for execution. A programmer not aware of this problem might write the following code segment for the *radar set module*:

```
awaitMbrtRanging( p );   // Wait until flight mode is ranging
RangeProcessing( p );    // Processing phase for ranging mode
```

A *range* event would cause any thread blocked on `awaitMbrtRanging()` to restart and become ready for execution. While in the ready state, a *track* event could occur. This would make the pending execution of `RangeProcessing()` unnecessary.

To avoid the *intercepted wakeup problem* the programmer could use the following code segment:

```
do
{
   awaitMbrtRanging( p );    // Wait until flight mode is ranging
}
while ( IsRanging( p ) == False );   // State inquiry operation

RangeProcessing( p );        // Processing phase for ranging mode
```

In this sequence, execution does not leave the `do while` loop until the *ranging condition* holds.

Listing 4.12 Update state transition event machine

```
 1 int UpdateMachine( STEOBJECT *p, int event )
 2 {
 3   Lock( &p→lock );
 4   FSM *fsm = p→fsm;
 5   int curState = p→state;
 6   int nextState = SelectNextState( fsm, curState, event );
 7
 8   UpdateEvtSemaphores( fsm, event );
 9   UpdateCallSemaphores( fsm, event );
10
11   UpdateCndSemaphores( fsm, nextState );
12   UpdateConditionTable( fsm );
13
14   UpdateMbrSemaphores( fsm, nextState );
15   UpdateSetSemaphores( fsm, nextState );
16   UpdateSetTable( fsm );
17
18   p→state = nextState;
19   UnLock( &p→lock );
20   return( 0 );
21 }
```

Synchronization Category	Trans[5]	Initialization
Unconditional Synchronization Operations		
Wait on Transition Event `awaitEvt`*`Relation(p)`*		`dop_updown(sema)`
Wait on Call `awaitCall`*`Relation(p)`*		`dop_updown(sema)`
Wait on Set Transition `awaitSett`*`Set(p)`*	@T @F	`dop_updown(stsema)` `dop_movesema(ltsema, stsema)`
Wait on Set Transition `awaitSetf`*`Set(p)`*	@T @F	`dop_updown(sfsema)` `dop_movesema(lfsema, sfsema)`
Conditional Synchronization Operations		
Wait on Condition `awaitCndt`*`Set(p)`*	@T @F	`dop_up(tsema)` `dop_down(tsema)`
Wait on Condition `awaitCndf`*`Set(p)`*	@T @F	`dop_down(fsema)` `dop_up(fsema)`
Wait on Set Membership `awaitMbrt`*`Set(p)`*	@T @F	`dop_up(tsema)` `dop_down(tsema)`
Wait on Set Membership `awaitMbrf`*`Set(p)`*	@T @F	`dop_down(fsema)` `dop_up(fsema)`

Table 4.5 Transition manager: semaphore update

4.4 Summary

The state transition event machine implementation provided by the `stetoc` compiler is incomplete. To provide a complete STEM implementation two additional

[5]Trans=transition. @T = false $\rightarrow$ true transition, @F = true $\rightarrow$ false transition.

components have been provided, namely, an implementation of the *d-operations* to provide synchronization services, and, a transition manager to access STE variables. Faulk and Parnas [3] have already specified that the *d-operations* described by Belpaire and Wilmotte [5] are sufficient for the synchronization requirements of state transition event machines. To provide these synchronization services, we extended the `syscall()` interface of an experimental microkernel to provide *d-operations* to user-mode applications.

The final component required to implement STEMs is a module we call the transition manager. This module is written in terms of the *d-operations* and data structures generated by the `stetoc` compiler. Transition manager handles all create, delete and update requests on an STE variable. Most importantly, transition manager contains the logic needed to manage the blocking and unblocking of executing threads.

Chapter 5

Experience and Results

In Chapters 3 and 4 we have described our implementation of the state transition event machine. In this chapter we would like to demonstrate the usability of STE variables and STE synchronization by examining the results taken from an application designed to verify our implementation.

5.1 Testing the STE Operations

We designed a simulator to verify the correct operation of the STE synchronization operations. This application is built around a core of threads which simulate the Radar Set Module environment described by Faulk and Parnas [3]. This state machine has been described in Section 3.1.1.

The simulation is event-driven. Simulation runs involve the input of a *state machine trace* which describes the sequence of events that are about to take place. The *state machine trace* is nothing more than a sequence of input characters coded

for specific events For example the following trace is a *canonical trace*[1] for the Radar Set Module

`arbatbasbatrsbrtstrsaq`

For this trace the events were coded in the following fashion r = range, s = standby, t = track, a = reliable, b = unreliable and q = quit

The output of this program is a log which summarizes all significant simulation events An extract from the simulation log of the *canonical trace* is shown below

```
471649    328    192    mm-start slice=7 clock=471702
473947    329    192    mm(Range) TRKREL->RNGREL restarts=18
474360    330    192    mm-end slice=8 clock=474429
474782    331    204    restart-0-awaitEvtRange 2
475080    332    204    pass-1-awaitEvtRange
475412    333    205    restart-0-awaitEvtRange 2
475716    334    205    pass-1-awaitEvtRange
476038    335    206    restart-0-awaitEvtRange 2
476334    336    206    pass-1-awaitEvtRange
```

The first column contains the time on the simulation clock This is followed by a record number and thread identifier The last column contains the thread response For example, record 329 logs the occurrence of a `range` event A more detailed log for this trace can be found in Appendix F 1 To analyze the simulation logs we developed an `awk` script which scans the log for operational violations

The simulator requires access to a number of programming interfaces These have been summarized in Table 5 2

[1] A *canonical trace* is the shortest trace which starts from the initial state of the state machine, visits all states and transitions, and returns to the initial state

5.1.1 Simulating the Radar Set Module

The test application is built around a core of five threads which simulate the Radar Set Module. The threads involved in the simulation are:

main: The main thread is responsible for creating all of the threads used in the simulation. At startup it creates the control monitor thread, then reads a *state machine trace* from a specified disk file. The information in the *state machine trace* is transferred to the control monitor. After the control monitor is started, main creates an instance of the *radar variable*.[2] The main monitor threads are then started (ModeMonitor, ReliabilityMonitor and FaultMonitor). main spawns a large number test threads (ForkEvtThreads, ForkCndThreads, ForkMbrThreads, ForkSetThreads) which immediately block on a number of STE semaphores associated with the STE variable. At this point the work of main is complete. main blocks on a thread mutex until the control monitor signals that the simulation is complete. At that point main prepares a log of simulation events.

ControlMonitor: This thread maintains four communication channels to main, ModeMonitor, ReliabilityMonitor, and FaultMonitor. When main initially connects with the control monitor, information from the *state machine trace* is transferred into an event queue. The primary job of the control monitor is to replay this event trace. The main monitors periodically request information from this event queue.

---

[2]A *radar variable* is an instance of the RADARVAR STE variable as described in Section 3.3.5.

`ModeMonitor`: This thread is setup as a periodic task which activates every 60 ms. When activated it polls the control monitor for the next event from the event queue. The mode monitor manages events of type `range`, `track`, and, `standby`. When these events are returned by the control monitor, the mode monitor uses the corresponding state transition operators (`Range`, `Track` and `Standby`) to force a transition in the state machine.

`ReliabilityMonitor`: This thread is setup as a periodic task which activates every 60 ms. When activated it polls the control monitor for the next event from the event queue. The reliability monitor manages events of type `reliable` and `unreliable`. When these events are returned by the control monitor, the reliability monitor uses the corresponding state transition operators (`Reliable` and `Unreliable`) to force a transition in the state machine.

`FaultMonitor`: This thread is setup as a periodic task which activates every 60 ms. When activated it polls the control monitor for the next event from the event queue. The fault monitor manages events of type `failure`. When this event is returned by the control monitor, the fault monitor executes the `Failure` transition operation to force a transition in the state machine. After the `Failure` operation is performed the fault monitor sends a request to the control monitor which terminates the simulation.

Taken together, these threads simulate the environment described by Faulk and Parnas [3]. The mode monitor, reliability monitor and fault monitor simulate periodic tasks with a period of about 60 ms. When these threads restart, they poll the control monitor for event information. If a new event has occurred, the

appropriate monitor will request a state machine update using a state transition operation. Execution of these state transition operations restart the test threads which are blocked on the corresponding STE semaphores. The test threads are then free to make appropriate annotations on the simulation log. The test threads are described in more detail in Section 5.1.2.

Our simulator was bound with the *transition manager* (see Section 4.2) and the radar set module generated by our precompiler (see Section 3.2.4). This executable was run under an extended version of the VSTa microkernel which provides services for the *d-operation* (see Section 4.2). In all simulation runs, the main monitors (control monitor, mode monitor, reliability monitor and failure monitor) were started in the highest priority scheduling queue (**PRI_RT**), while the test threads were started in the medium priority scheduling queue (**PRI_TIMESHARE**).

5.1.2 Test Threads

We designed four types of threads to test our STE synchronization operations.

EVT Test Threads: These threads were used to test the *Wait on Transition* synchronization operations. We provided threads to test the `range`, `track`, `standby`, `reliable` and `unreliable` forms of this operator.

CND Test Threads: These threads were used to test the *Wait on Condition* synchronization operations. We provided threads to test one condition for each relation attribute.

MBR Test Threads: These threads were used to test the *Wait on Set Membership* synchronization operations. We provided threads to test the sets

`Tracking`, `Ranging`, and `Isstandby`.

SET Test Threads. These threads were used to test the *Wait on Set Transition* synchronization operations. We provided threads to test the sets `Tracking` and `IsReliable`.

Test threads were designed with the following goals in mind:

- Threads should always be able to block on some STE semaphore.
- Threads must record state information on a simulation log immediately before passing onto a STE semaphore and after any restarts.
- Where assertions have been provided, threads must execute assertions immediately after they restart.

In our simulations the main thread creates 42 test threads at startup. Depending on the exact state of the STEM, as many as 21 threads can be restarted when the *radar variable* is updated.

5.1.2.1 EVT Test Thread

An EVT test thread is used to test a *Wait on Transition* synchronization operation (see Section 3.4.7.1). Listing 5.1 shows the code for a test thread testing the `awaitEvtTrack` synchronization operation. This thread can only be awakened when the mode monitor executes the `Track` state transition operation.

Listing 5.1 EvtTrack: a thread to test awaitEvtTrack.

```
1: void EvtTrack( RADARVAR *RS )
2: {
3:   int i, res;
4:
5:   for ( i = 0; Running(); ++i ) do
6:     if ( DEBUG(16) ) then
7:       print_it( 0, "pass-%ld: awaitEvtTrack" );
8:     end if
9:
10:     res = awaitEvtTrack( RS );
11:
12:     if ( DEBUG(16) ) then
13:       print_it( 0, "restart-%ld: awaitEvtTrack: %d", i, res );
14:     end if
15:   end for
16:
17:   mutex_thread( rootproc );
18:   ThreadExit();
19: }
```

5.1.2.2 CND Test Thread

A CND test thread is used to test *Wait on Condition* synchronization operations (see Section 3.4.7.4). Listing 5.2 shows the code for a thread testing the awaitCndtTrack and awaitCndfTrack synchronization operations. The goal of CndTrack is to keep the executing thread blocked.

To test the *radar variable's* condition, we use the predicate IsTrack(RS,TRKREL). If the condition holds, the thread will pass onto awaitCndfTrack and block until the condition does not hold. If the condition does not hold, then the thread will pass onto awaitCndtTrack and block until the condition does hold. If a CND test thread is restarted under the wrong condition, it will create an appropriate

annotation in the simulation log

5.1.2.3 MBR Test Thread

A MBR test thread is used to test *Wait on Set Membership* synchronization operations (see Section 3.4.7.5). Listing 5.3 shows the code for a thread testing the `awaitMbrfTracking`, `awaitMbrfRanging` and `awaitMbrfIsstandby` synchronization operations. The goal of `MbrfMode` is to keep the executing thread blocked. The `IsRanging`, `IsTracking` and `IsIsstandby` inquiry operations are used to determine the current mode of the state machine. With this information the thread selects the corresponding synchronization operation and blocks until the mode is changed by the mode monitor. If a MBR test thread is restarted in an unexpected mode, it will create an appropriate annotation in the simulation log.

Listing 5.2 CndTrack a thread to test awaitCndTrack

```
1  void CndTrack( RADARVAR *RS )
2  {
3    int i, res;
4    Boolean cond;
5
6    for ( i = 0; Running() ; ++i ) do
7      if ( IsTrack( RS, (RADARVAR *)TRKREL ) == False ) then
8        if ( DEBUG(32) ) then
9          print_it( 0, "pass-F-%ld: awaitCndtTrack" );
10       end if
11
12       res = awaitCndtTrack( RS, (RADARVAR*)TRKREL );
13       cond = IsTrack(RS,(RADARVAR *)TRKREL);
14       ASSERTION( cond==True );
15
16       if ( DEBUG(32) ) then
17         print_it(0, "restart-F-%ld: awaitCndtTrack: %ld", i, res);
18       end if
19     else
20       if ( DEBUG(32) ) then
21         print_it(0, "pass-T-%ld: awaitCndfTrack" );
22       end if
23
24       res = awaitCndfTrack( RS, (RADARVAR*)TRKREL );
25       cond = IsTrack(RS,(RADARVAR *)TRKREL);
26       ASSERTION( cond==False );
27
28       if ( DEBUG(32) ) then
29         print_it(0, "restart-T-%ld: awaitCndfTrack: %ld", i, res);
30       end if
31     end if
32   end for
33
34   mutex_thread( rootproc );
35   ThreadExit();
36 }
```

Listing 5.3 MbrfMode: a thread to test awaitMbrtMode.

```
1  void MbrfMode( RADARVAR *RS )
2  {
3    int i, res;
4    Boolean cond;
5
6    for ( i = 0; Running(); ++i ) do
7      if ( IsRanging( RS ) == True ) then
8        if ( DEBUG(64) ) then
9          print_it(0, "pass-T-%ld: awaitMbrfRanging" );
10       end if
11       res = awaitMbrfRanging( RS );
12       cond = IsRanging( RS ); ASSERTION( cond==False );
13       if ( DEBUG(64) ) then
14         print_it(0, "restart-T-%ld: awaitMbrfRanging: %d", i, res );
15       end if
16
17     else if ( IsIsstandby( RS ) == True ) then
18       if ( DEBUG(64) ) then
19         print_it(0, "pass-T-%ld: awaitMbrfIsstandby" );
20       end if
21       res = awaitMbrfIsstandby( RS );
22       cond = IsIsstandby( RS ); ASSERTION( cond==False );
23       if ( DEBUG(64) ) then
24         print_it(0, "restart-T-%ld: awaitMbrfIsstandby: %d", i, res);
25       end if
26
27     else if ( IsTracking( RS ) == True ) then
28       if ( DEBUG(64) ) then
29         print_it(0, "pass-T-%ld: awaitMbrfTracking" );
30       end if
31       res = awaitMbrfTracking( RS );
32       cond = IsTracking( RS ); ASSERTION( cond==False );
33       if DEBUG(64) ) then
34         print_it(0, "restart-T-%ld: awaitMbrfTracking: %d", i, res);
35       end if
36     end if
37   end for
38
39   mutex_thread( rootproc );
40   ThreadExit();
41 }
```

5.1.2.4 SET Test Thread

A SET test thread is used to test *Wait on Set Transition* synchronization operations (see Section 3.4.7.3). Listings 5.4 and 5.5 show the code for a thread testing the `awaitSettTracking` and `awaitSetfTracking` synchronization operations.

The goal of `SetTracking` is to keep the executing thread blocked. The `IsTracking` inquiry operation divides the test into two parts:

- Cases where the starting condition is `True` (this includes the possibilities `awaitSetfTracking(tf)` and `awaitSettTracking(tft)`), and,
- Cases where the starting condition is `False` (this includes the possibilities `awaitSettTracking(ft)` and `awaitSetfTracking(ftf)`.

Within each case we select a candidate operation by consulting the for-loop index. If the index is even, we select the one-transition form of the synchronization operation. If the index is odd, we select the two-transition form of the synchronization operation. The test thread described here, `SetTracking`, will be restarted by the mode monitor after the correct number of `track` transitions have occurred.

Listing 5.4 SetTracking: a thread to test awaitSetTracking.

```
1  void SetTracking( RADARVAR *RS )
2  {
3    int i, res;
4
5    for ( i = 0; Running() ; ++i ) do
6      if ( IsTracking( RS ) == True ) then
7
8        if ( i % 2 == 0 ) then
9          tf_sequence (see lines 1--8 in Listing 5.5)
10       else
11         tft_sequence (see lines 10--17 in Listing 5.5)
12       end if
13
14     else
15       if ( i % 2 == 0 ) then
16         ft_sequence (see lines 19--26 in Listing 5.5)
17       else
18         ftf_sequence (see lines 28--35 in Listing 5.5)
19       end if
20     end if
21   end for
22
23   mutex_thread( rootproc );
24   ThreadExit();
25 }
```

Listing 5.5 Set tests for `awaitSetTracking`

```
1  // tf-sequence
2
3  if ( DEBUG(128) ) then
4    print_it(0, "pass-T-%ld: T->F: SetfTracking" );
5  end if
6  res = awaitSetfTracking( RS );
7  if ( DEBUG(128) ) then
8    print_it(0, "restart-T-%ld: T->F: SetfTracking: %ld", i, res);
9  end if
10
11 // tft-sequence
12
13 if ( DEBUG(128) ) then
14   print_it(0, "pass-T-%ld: T->F->T: SettTracking" );
15 end if
16 res = awaitSettTracking( RS );
17 if ( DEBUG(128) ) then
18   print_it(0, "restart-T-%ld: T->F->T: SettTracking: %ld", i, res);
19 end if
20
21 // ft-sequence
22
23 if ( DEBUG(128) ) then
24   print_it(0, "pass-F-%ld: F->T: SettTracking" );
25 end if
26 res = awaitSettTracking( RS );
27 if ( DEBUG(128) ) then
28   print_it(0, "restart-F-%ld: F->T: SettTracking: %ld", i, res);
29 end if
30
31 // ftf-sequence
32
33 if ( DEBUG(128) ) then
34   print_it(0, "pass-F-%ld: F->T->F: SetfTracking" );
35 end if
36 res = awaitSetfTracking( RS );
37 if ( DEBUG(128) ) then
38   print_it(0, "restart-F-%ld: F->T->F: SetfTracking: %ld", i, res);
39 end if
```

5.1.3 Simulation Results

The `awk` script used to verify the simulation logs generated by the Radar Set Module (RSM) simulator maintains a RSM state machine. When it reads a record generated by an event-signaling process it performs the next state calculation and compares it's result with the corresponding result embedded in the simulation record. If the next state in the simulation record differs from the verifier's calculation, we say there has been an operational failure in the RSM simulator. After each event is verified, a number of thread restart records must be verified. As the verifier processes restart records it compares it's internal state with the information in the simulation record. Any difference between it's internal state and the information in the simulation record indicates an operational failure of the RSM simulator. The verifier processes four restart conditions:

EVT thread restart: When the verifier reads a 'restart `awaitEvtRelation`' record, `Relation` must be equal to the relation recorded by the verifier during the most recent next state calculation.

CND thread restart: When the verifier reads a 'restart `awaitCndRelation`' record, the transition $(S \rightarrow S')$ must be a member of `Relation` and $(S \rightarrow S')$ must be embedded in the simulation record. Here S is the current state of the verifier and S' is some state reachable from S in one transition.

MBR thread restart: When the verifier reads a 'restart `awaitMbrSet`' record, the `Set` embedded in the simulation record must be equal to the last recorded *mode* in the state machine maintained by the verifier.

SET thread restart: When the verifier reads a 'restart `awaitSetSet`' record, the `Set` and predicate state embedded in the simulation record must be equal to the current mode and predicate state in the state machine maintained by the verifier. The verifier also ensures that the correct number of set transitions have occurred before the restart.

Several *state machine traces* of varying lengths were processed using our simulator. The results of the verification step on these simulation logs are presented in Appendix F.2 and Table 5.1. In all cases the verification script found no operational failures in the simulation logs. We conclude that our implementation of the STE synchronization operations satisfies the descriptions given in Faulk and Parnas [3].

Run Name	Length of Simulation (seconds)	Number of Events	Number of Operations Checked	Number of Verification Failures
canon	1.19	21	303	0
canon20	9.29	200	3203	0
random10	3.62	99	1562	0
random20	7.06	198	3201	0

Table 5.1: Simulation results.

Table 5 2 Interfaces utilized by the test application

Function	Description
In-core Logging : icl.h	
icl_initialize	Initialize in-core logging
icl_destroy	Reset in-core logging
icl_insert	Add a line to the in-core log
icl_retrieve	Retrieve a line from the in-core log
icl_writelog	Write in-core log to the simulation log
Simulation Log : logfile h	
logf_initialize	Initialize the simulation log
logf_write	Write a line to the simulation log
Formatted Print : print h	
print_initialize	Declare print sink for `print_it`
print_it	Format output to the print sink
Event Queue : queue.h	
queue_create	Create an event queue
queue_destroy	Destroy an event queue
queue_add	Add an element to the event queue
queue_remove	Remove an element from event queue
queue_front	Check the front element of event queue
queue_empty	Test if event queue is empty
Communications Monitor : cproc c	

continued on next page

continued from previous page

Function	Description
comm_createServer	Convert thread to communications server
comm_connectServer	Connect to the communications server
comm_readBuffer	Get next event from communications server
control_read	Process FS_READ message
control_write	Process FS_WRITE message
VSTa functions : syscalls.h	
tfork	Create a thread
gettid	Return a thread's thread identifier
__msleep	Sleep for n milliseconds
mutex_thread	Implements barrier synchronization
msg_connect	Connect to a server at a known port
msg_accept	Accept a client connection
msg_disconnect	Client disconnects from server
msg_send	Send a message
msg_receive	Receive a message
Miscellaneous functions : ktest.h	
LoadEventsFile	Read a state-machine trace from a file
SignalFailure	Report failed state transition operator
Running	Predicate indicating if simulation is running
DEBUG	Macro controlling output to log
ASSERTION	Macro used to expand assertions

continued on next page

continued from previous page	
Function	**Description**
GetThreadPriority	Get caller's scheduling queue
SetThreadPriority	Set caller's scheduling queue
ThreadExit	Terminate a thread
Radar Set Module : radar.h	
CreateRadar	Create a radar STE variable
DeleteRadar	Delete a radar STE variable
IsRanging, IsTracking, IsIsstandby, IsNotReliable, IsIsreliable	Relation inquiry operations
Range, Track, Standby, Unreliable, Reliable, Failure	State transition operations
awaitEvt*Relation*	Wait on transition operation
awaitCnd*Relation*	Wait on condition operation
awaitMbr*Set*	Wait on set membership operation
awaitSet*Set*	Wait on set transition operation

5.2 Timing Characteristics of Synchronization Operations

In order to implement many real-time applications it is necessary to measure the timing characteristics of executing processes. To this end, we measured the timing characteristics of a number of the low-level and high-level operations we have presented in Chapters 3 and 4. All measurements were performed on a single i486/33MHz IBM compatible computer running release 1.5.2 of the VSTa operating system.

VSTa provides direct access to its time-of-day clock via the **time_get** system call. **time_get** returns the value of the time-of-day clock in microseconds. To make timing measurements on simple operations, we used the code segment illustrated in Listing 5.6. **cvtime** returns the difference between two timestamps in microseconds. Since the clock is running for part of each **time_get** system call, the value returned by this code segment is an overestimate of elapsed time. We adjusted all timing measurements made in this fashion by subtracting the time it takes to execute a single **time_get** function. The median execution time for **time_get** is 27 μs (see Table 5.3).

A simple application was designed to collect the timing measurements given in Table 5.3. This application performs the measurements and outputs the results to an ASCII data file. These results are then collected and input into SPSS[3] to summarize the timing measurements.

---

[3]We used SPSS for Windows Release 7.5.1. SPSS is a trademark of SPSS Inc.

Listing 5.6 Measuring the time to execute an operation.

```
1  int usecs;
2  sid_t sema = 0;
3  struct time start, end;
4
5  time_get( &start );
6  sema = dop_create( S_CLOSED );
7  time_get( &end );
8  usecs = cvtime( &end, &start );
```

For some calculations it is necessary to know the number of STE semaphores associated with a STE variable. For our implementation of STE variables we calculate the number of semaphores using the following relation:

$$n = 2n_r + 6n_s + 2n_c$$

where n is the number of semaphores, n_r is the number of relation attributes in the state machine definition, n_s is the number of set attributes in the state machine definition, and, n_c is the number of conditions identified by the `stetoc` precompiler.

Since the state machine definition for *radar variables* includes 6 relation attributes, 5 set attributes and 13 conditions, 68 STE semaphores are required to implement *radar variables*.

5.2.1 dop_create and dop_destroy

Creating a STE semaphore (`dop_create`) has a median execution time of about 35 μs. The timing measurements for the `dop_destroy` operation are a bit more interesting. For the reported measurements, we destroyed 255 STE semaphores. The first destroy operation took 176 μs, the last destroy operation took 769 μs.

Fitting a curve to these 255 measurements yields the relation

$$y = 2.3217x + 151.0618, r^2 = 0.995$$

where x = the number of semaphores destroyed and y = the time to destroy x semaphores, in microseconds. The coefficient of determination indicates that 99.5% of the variation in the y variable is explained by the fitted regression.

We explain this behaviour in the following fashion. In VSTa , the debug version of the kernel's memory allocation module checks for instances of *double-freeing* in the `free` function. So as more items are freed (in this case memory allocated to VSTa semaphores) the kernel must review its ever increasing list of freed objects to insure that candidate objects are not being freed twice. What we are actually measuring in this case is an artifact in the kernels memory allocation algorithm.

5.2.2 Wait-Wake Response

This is perhaps the single most important timing characteristic for any synchronization method which employs blocking. Measuring wait-wake response answers the question: *'How quickly can a blocked thread respond to a signal from another thread ?'* To measure this response time we used the code shown in Listing 5.7.

The execution sequence of the two threads T_1 and T_2 goes something like this:

1. Thread T_1 executes the passage operation on semaphore A and blocks.

2. Thread T_2 executes the signal operation on semaphore A. This restarts thread T_1.

3. Thread T_2 executes the passage operation on semaphore B and blocks.

4. Thread T_1 executes the signal operation on semaphore B. This restarts thread T_2.

5. Return to step 1.

From this perspective we see that the measurement of wait-wake response gives us a very good idea of the context switching time for threads running in the VSTa operating system. The four steps given above represent two context switches: T_1 to T_2 then T_2 to T_1. In Table 5.3 the median execution time for the wait-wake response is 142 μs. This implies that the context switching time in VSTa is about 71 μs.

In Section 4.2 we discussed two approaches for the design of our low level synchronization operations (d-operations). With one approach we could provide semaphore services by designing a VSTa server. With this approach the only means of interprocess communications would be the native messaging system. In Table 5.3 we provide measurements for VSTa's **msg_send** system call. For 0-byte messages the **msg_send** system call has a median response time of 274 μs. These response times are significantly lower than those reported here for wait-wake response. This reinforces our comments in Section 4.2 which suggest that extending the microkernel to provide semaphores results in lower latency semaphore operations.

Listing 5.7 Measuring wait-wake response

```
1  sid_t asema = 0;
2  sid_t bsema = 0;
3  int n = number_of_samples;
4
5  void main()
6  {
7    asema = dop_create( S_CLOSED );
8    bsema = dop_create( S_CLOSED );
9    tfork( T1, 0 );
10   tfork( T2, 0 );
11   ...
12 }
13
14 void T1( int arg )
15 {
16   struct time start[n], end[n];
17
18   for ( int i= 0; i < n ; ++i ) do
19     time_get( &start[i] );
20     dop_pass( asema );
21     time_get( &end[i] );
22     dop_updown( bsema );
23   end for
24 }
25
26 void T2( int arg )
27 {
28   struct time start[n], end[n];
29
30   for ( int i= 0; i < n ; ++i ) do
31     dop_updown( asema );
32     time_get( &start[i] );
33     dop_pass( bsema );
34     time_get( &end[i] );
35   end for
36 }
```

5.2.3 dop_up and dop_movesema

Timing measurements for the **dop_up** and **dop_movesema** operations characterize the overhead associated with restarting and morphing threads. To measure the median restart time we used the algorithm presented in Listing 5.8. In the runs reported in Table 5.3 we recorded the time to restart 128 threads as 1228 μs. This leads to a median restart time of about 10 μs per thread. This is the time required to move one thread from a semaphore queue to some scheduling queue.

Listing 5.8 Measuring median restart time.

```
1:  void main()
2:  {
3:    int n = number_of_samples;
4:    int usecs, restarts;
5:    struct time start, end;
6:    sid_t sema = dop_create( S_CLOSED );
7:
8:    for ( int i = 0 ; i < n ; ++i ) do
9:      tfork( thread, sema );
10:    end for
11:
12:    time_get( &start );
13:    restarts = dop_up( sema );
14:    time_get( &end );
15:    assert( restarts == n );
16:    usecs = cvtime( &end, &start );
17:  }
18:
19:  void thread( sid_t sema )
20:  {
21:    dop_pass( sema );
22:  }
```

To measure the median morphing time we used the algorithm presented in

Listing 5.9. In the runs reported in Table 5.3 we recorded the time to morph 128 threads as 442 μs. This leads to a median morphing time of about 4 μs per thread. This is the time required to move one thread from a source semaphore queue to a destination semaphore queue.

Listing 5.9 Measuring median morphing time.

```
1  void main()
2  {
3    int n = number_of_samples;
4    int usecs, restarts, movers;
5    struct time start, end;
6    sid_t asema = dop_create( S_CLOSED );
7    sid_t bsema = dop_create( S_CLOSED );
8
9    for ( int i = 0 ; i < n ; ++i ) do
10     tfork( thread, asema );
11   end for
12
13   time_get( &start );
14   movers = dop_movesema( asema, bsema );
15   time_get( &end );
16   usecs = cvtime( &end, &start );
17
18   assert( movers == n );
19   restarts = dop_up( asema );
20   assert( restarts == 0 );
21   restarts = dop_up( bsema );
22   assert( restarts == n );
23 }
24
25 void thread( sid_t sema )
26 {
27   dop_pass( sema );
28 }
```

5.2.4 High-Level Synchronization Operations

We measured the times to create, delete and update *radar variables* by instrumenting the simulator described in Section 5.1. During typical simulation runs, 42 test threads are created by the `main` thread, and up to 21 threads may be restarted when a *radar variable* is updated.

The results presented in Table 5.3 indicate that the median creation time for a *radar variable* is about 2.5 ms. Almost all of this time can be accounted for by the creation of the STE semaphores. In Section 5.2.1 we reported a median semaphore creation time of 35 μs. Since *radar variables* contain 68 STE semaphores, the `dop_create` operation accounts for approximately 98% (2380 μs) of the *radar variable* creation time. The remaining time is required to initialize the set and condition tables of the STEM.

The median execution time for the `DeleteRadar` operation is about 15.7 ms. Taking into account the behaviour of the `dop_destroy` operation reported in Section 5.2.1, we can account for 1820 μs (12%) of this time (time to restart 42 threads + time to destroy 68 semaphores + time to free 14 memory blocks).

The median execution time for the `UpdateRadar` operation is about 941 μs. After examining the results in Section 5.2.3 and knowing that up to 21 threads are restarted with each transition event, it is clear that only about 200 μs (20%) of the update cycle can be attributed to restarting threads. The remaining time is required for scanning and updating the *radar variable's* set and condition tables.

Operation	Median μs	Sample Size	Range μs
Low-Level Operations			
`dop_create`	35	255	31–56
`dop_destroy`	††	255	176–769
`wait-wake response`	142	255	139–177
`dop_up` restart 128 threads	1223	126	1203–1466
`dop_movesema` move 128 threads	442	126	423–549
High-Level Operations			
`CreateRadar`	2468	127	2458–2526
`DeleteRadar`	15691	127	15416–16017
`UpdateRadar`	941	127	795–1030
VSTa Operations			
`time_get`	27	510	25–29
`tfork`	444	80	435–457
`msg_send` 0-byte message	274	126	271–300
`msg_send` 64-byte message	921	126	919–962
`malloc`	127	127	119–677
`free`	79	127	77–81

Table 5.3: Timing characteristics of STE operations

††See Section 5.2.1 for an explanation

5.3 Summary

We have demonstrated that our implementation of the state transition event machine conforms to the specification outlined in Faulk and Parnas [3]. This was accomplished by designing a simple test application which forces threads to block on specific STE semaphores. These threads record their response to state transitions to a simulation log which is later analyzed by a verification program. In the simulation runs described here, no violations were reported by our verification program.

In this chapter we have also presented the timing characteristics of several important low-level and high-level STEM operations. We measured the *wait-wake response* of the *d-operations* to be about 142 μs. The transition manager was able to update the *radar* STEM in about 941 μs. These results suggest that even under a moderate load, the transition manager should be able to process several hundred updates per second.

Chapter 6

Conclusion

The principal goal of this thesis was to provide an implementation for state transition event machines (STEM). Our approach in the design and implementation of STEMs was:

- Develop a tool to automate conversion of STE type declarations into compilable 'C' code.
- Provide synchronization support for state transition event machines.
- Develop a technique to verify the correct operation of a STEM.
- Examine the execution performance of our implementation.

In this chapter we state our findings and describe possible future work.

6.1 Contributions and Achievements

We have succeeded in our main goal of providing a usable implementation of state transition event machines. The tool we designed to convert STE type declarations to compilable 'C' code works very well. One discovery we made, however, was that the specification given in [3] only provides STE operations for blocking tasks. Faulk and Parnas [3, 4] discuss in very general terms how the *d-operations* can be used to effect a restart of blocked tasks, but a more substantial model was required to manage state transition event machines. We addressed this shortcoming by designing and implementing an abstract data type to provide these management operations. This object is referred to as a transition manager.

To provide the synchronization support required by STEMs, we extended the system call interface of the VSTa operating system to include our implementation of the *d-operations* [5, 3].

We use an event-driven simulation approach to check for the correct operation of specific STEMs. During a simulation a group of test threads respond to events injected into the system by recording their response in the simulation log. After the simulation, this log is checked with a verification tool which identifies suspect operations. Several simulations of varying lengths were run using this approach. In all cases the verification tool found no errors in the simulation logs. We conclude that our implementation of the STE synchronization operations satisfy the descriptions given in Faulk and Parnas [3].

We also examined the execution performance of the *radar* STEM running in the VSTa environment. The measured *wait-wake response time* for the *d-operations* was about 142 μs. This indicates that state transition event machines can be

fairly responsive to input signals. The transition manager is able to perform a STEM update in about 941 μs. This indicates that even under a modest load, the transition manager can perform several hundred updates per second.

Our approach to state transition event machines appears to be the first published implementation of the STEM abstract data type. Our approach should be usable in any event-driven concurrent system which can be specified as a finite state machine. Once we understand the nature and specification of an event-driven state machine, we can provide a tool, `stetoc`, which can convert the STE type declaration directly into a compilable representation of the STEM. Since a STE declaration can imply hundreds of STE operations, this tool should allow programmers to design their applications without the necessity of manually coding each of the STE operations.

6.2 Future Work

The tools and techniques described here could be very useful in the construction of embedded systems based on STEM prototypes. However, we feel that the automation tool (precompiler) should be enhanced to include the generation of the code necessary to verify the state transition event machine.

At present our automation tool only handles simple events of the form:

```
@T(condition)
```

A future extension should handle more general events of the form:

```
@T(condition1) when (condition2)
```

which indicates that the event specified in `condition1` will not be recognized unless `condition2` holds.

At present our implementation and approach has been validated in a concurrent environment which is best described as a simple reactive system. Future work should test this implementation in hard and soft real-time systems.

References

[1] Constance Heitmeyer and Dino Mandrioli. Formal methods for real-time computing: An overview. In Constance Heitmeyer and Dino Mandrioli, editors, *Formal Methods for Real-Time Computing*, number 5 in Trends in Software, pages 1–32. John Wiley and Sons, Chichester, England, 1996.

[2] Stuart R. Faulk and David L. Parnas. On the uses of synchronization in hard-real-time systems. Unpublished technical report, October 1983.

[3] Stuart R. Faulk and David L. Parnas. On synchronization in hard-real-time systems. *Communications of the ACM*, 31(3):274–287, March 1988.

[4] Stuart R. Faulk. *State Determination in hard-embedded systems*. PhD dissertation, University of North Carolina, Chapel Hill, 1989.

[5] G. Belpaire and J.P. Wilmotte. A semantic approach to the theory of parallel processes. In *Proceedings of the 1973 European ACM Symposium*, pages 159–164. ACM, 1973.

[6] Ron Koymans, Ruurd Kuiper, and Erik Zijlstra. Paradigms for real-time systems. In G. Goos and J. Hartmanis, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, number 331 in LNCS, pages 159–174. Springer-Verlag, New York, 1988.

[7] E.W. Dijkstra. *Cooperating sequential processes*, pages 43–112. Programming Languages. Academic Press, New York, 1968.

[8] C. R. Snow. *Concurrent Programming*. Cambridge Computer Science Texts. Cambridge University Press, Cambridge, England, 1992.

[9] B.O. Gallmeister. *POSIX.4: Programming for the Real World*. O'Reilly and Associates, Inc., Sebastopol, 1995.

[10] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Professional Computing Series. Addison Wesley Longman, Inc., 1997.

[11] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[12] P. Brinch Hansen. *Operating System Principles*. Prentice-Hall, 1973.

[13] J.J. Horning A.D. Birrell, J.V. Guttag and R. Levin. Synchronization primitives for a multiprocessor: A formal specification. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 94–102, Palo Alto, 1987. ACM.

[14] Anthony LaMarca. A performance evaluation of lock-free synchronization protocols. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 130–140, New York, 1994. ACM.

[15] Thomas E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[16] Joanne M. Atlee and John Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

[17] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the theory of computation*, chapter 2.1, pages 49–54. Prentice-Hall Inc., Englewood Cliffs, New Jersey, second edition, 1981.

[18] Andrew Valencia. An overview of the vsta microkernel. Technical report, Valencia Consulting, 1996. http://www.zendo.com/vsta/vsta_intro.html.

[19] Marc Donner, David Jameson, and William Moran Jr. Events: A structuring mechanism for a real-time runtime system. In *Proceedings of the Real-Time Systems Symposium*, pages 288–299, Los Alamitos, 1989. The IEEE, Inc.

Appendix A

Sample STE Type Declarations

A.1 Type Declaration for Radar Set Module

```
1   # Author:   Gordon O'Connell: Adapted from Faulk and Parnas,1988
2   # File:     radar.ste
3   # Date:     30 May 1997
4   #
5   # Radar Set Module: Control a virtual radar device.
6
7   ste radar
8   {
9       state
10      {
11          trkrel, rngrel, stbyrel, trknot, rngnot, stbynot, failed
12      },
13
14      relation reliable
15      {
16          (rngnot -> rngrel), (trknot -> trkrel), (stbynot->stbyrel)
17      },
18
19      relation unreliable
20      {
21          (rngrel ->rngnot), (trkrel -> trknot), (stbyrel -> stbynot)
22      },
23
24      relation track
```

```
25        {
26            (rngnot -> trknot),  (rngrel -> trkrel),
27            (stbynot -> trknot), (stbyrel -> trkrel)
28        },
29
30        relation range
31        {
32            (trknot -> rngnot),  (trkrel -> rngrel),
33            (stbynot -> rngnot), (stbyrel -> rngrel)
34        },
35
36        relation standby
37        {
38            (trknot -> stbynot), (trkrel -> stbyrel),
39            (rngnot -> stbynot), (rngrel -> stbyrel)
40        },
41
42        relation failure
43        {
44            (trkrel -> failed), (rngrel -> failed), (stbyrel -> failed),
45            (trknot -> failed), (rngnot -> failed), (stbynot -> failed)
46        },
47
48        set ranging
49        {
50            rngrel, rngnot
51        },
52
53        set isstandby
54        {
55            stbyrel, stbynot
56        },
57
58        set isreliable
59        {
60            rngrel, trkrel, stbyrel
61        },
62
63        set notreliable
64        {
65            rngnot, trknot, stbynot
66        },
67
68        set tracking
69        {
70            trkrel, trknot
```

```
71         },
72     },
```

A.2 Type Declaration for Mail Manager Module

```
1     # Author:   Gordon O'Connell  Adapted from Faulk and Parnas,1983
2     # File:     radar ste
3     # Date:     30 May 1997
4     #
5     # Mail Manager: Control email dispatch.
6
7     ste mail
8     {
9         state
10        {
11            rdynil, nrdynil, rdymsg, nrdymsg, callreq, calling, failed
12        },
13
14        relation enable
15        {
16            (nrdynil->rdynil), (nrdymsg->rdymsg)
17        },
18
19        relation msgarrival
20        {
21            (nrdynil->nrdymsg), (rdynil->rdymsg),
22            (nrdymsg->nrdymsg), (rdymsg->rdymsg)
23        },
24
25        relation requestcall
26        {
27            (nrdynil->callreq), (rdynil->callreq), (nrdymsg->callreq),
28            (rdymsg->callreq), (callreq->callreq)
29        },
30
31        relation disable
32        {
33            (rdynil->nrdynil), (rdymsg->nrdymsg)
34        },
35
36        relation callinit
37        {
38            (callreq->calling), (rdymsg->calling)
39        },
```

```
40
41      relation failure
42      {
43          (rdynil->failed), (nrdynil->failed), (rdymsg->failed),
44          (nrdymsg->failed), (callreq->failed), (calling->failed)
45      },
46  },
```

Appendix B

Scanner and Parser Description Files

B.1 Scanner Description File

```
1    %{
2    /* File:         ste.l
3     * Author:       Gordon O'Connell
4     * Date:         30 May 1997
5     *
6     * Scanner for STE Definition files. Scanner also provides line numbering
7     * and simple comments (#).
8     */
9
10   #include "parser.h"
11   #include "lexyy.h"
12   int lineno = 1;
13   %}
14
15   delim        [ \t]
16   ws           {delim}+
17   letter       [A-Za-z]
18   digit        [0-9]
19   identifier   ({letter}({letter}|{digit}|_)*)
20   eol          \n
```

```
21  comment      #.*
22
23  %%
24
25  {ws}              {}
26  {comment}         {}
27  ste               { return( YSTE ); }
28  state             { return( YSTATE ); }
29  set               { return( YSET ); }
30  relation          { return( YRELATION ); }
31  {identifier}      { yylval.stringval = yytext; return( YIDENTIFIER ); }
32  "->"              { return( YARROW ); }
33  ","               { return( ',' ); }
34  ";"               { return( ';' ); }
35  "("               { return( '(' ); }
36  ")"               { return( ')' ); }
37  "{"               { return( '{' ); }
38  "}"               { return( '}' ); }
39  {eol}             { lineno++; }
40  .                 { fprintf( yyout, "unrecognized character %c\n",
41                        *yytext ); }
42
43  %%
44
45  /*
46   * Default Action: scanner returns a zero token to report the end
47   * of file.
48   */
49
50  int yywrap( void )
51  {
52      return( 1 );
53  }
```

B.2 Parser Description File

```
1    %{
2    /* File:          ste.y
3     * Author:        Gordon O'Connell
4     * Date:          30 May 1997
5     *
6     * Parser for STE Definition files
7     */
8
9    #include <stdio.h>
10   #include <stdlib.h>
11
12   #include "lexyy.h"
13   #include "cons.h"
14   #include "deftoc.h"
15   #include "parser.h"
16   %}
17
18   %token YSTE
19   %token YSTATE
20   %token YSET
21   %token YRELATION
22   %token YIDENTIFIER
23   %token YARROW
24
25   %%
26
27   File:
28       YSTE Identifier '{' Declarations '}' ';'
29       { SyntaxTree = $$.node = SteType( Declarations( $4.list ),
30                                         $2.node) ; }
31       ;
32
33   Declarations:
34       { $$.list = NILLIST; }
35       | DeclarationList
36       { $$.list = $1.list; }
37       ;
38
39   DeclarationList:
40       OneDecl
41       { $$.list = list1( $1.node ); }
42       | DeclarationList OneDecl
43       { $$.list = cons( $2.node, $1.list ); }
44       ;
```

```
45
46  OneDecl:
47      StateDecl
48      { $$.node = $1.node; }
49      | RelationDecl
50      { $$.node = $1.node; }
51      | SetDecl
52      { $$.node = $1.node; }
53      ;
54
55  StateDecl:
56      YSTATE '{' IdentifierList '}' ';'
57      { $$.node = StateDecl( IdentifierList( $3.list )); }
58      ;
59
60  RelationDecl:
61      YRELATION Identifier '{' OrderedPairs '}' ';'
62      { $$.node = RelationDecl( OrderedPairs( $4.list ), $2.node ); }
63      ;
64
65  SetDecl:
66      YSET Identifier '{' IdentifierList '}' ';'
67      { $$.node = SetDecl( IdentifierList( $4.list ), $2.node ); }
68      ;
69
70  OrderedPairs:
71      OrderedPair
72      { $$.list = list1( $1.node ); }
73      | OrderedPairs ',' OrderedPair
74      { $$.list = cons( $3.node, $1.list ); }
75      ;
76
77  OrderedPair:
78      '(' Identifier YARROW Identifier ')'
79      { $$.node = OrderedPair( $2.node, $4.node ); }
80      ;
81
82  IdentifierList:
83      Identifier
84      { $$.list = list1( $1.node ); }
85      | IdentifierList ',' Identifier
86      { $$.list = cons( $3.node, $1.list ); }
87      ;
88
89  Identifier:
90      YIDENTIFIER
```

```
91        { $$.node = Identifier( $1.stringval ); }
92        ;
93
94    %%
95
96    /*
97     * Parser syntax error reporter. Syntax errors are fatal.
98     */
99
100   void yyerror( char *string )
101   {
102      extern int lineno;
103      static char fmtbuf[128];
104
105      sprintf( fmtbuf, "Line: %d, %s", lineno, string );
106      panic( fmtbuf );
107   }
```

Appendix C

Sample Precompiler Output

C.1 Interface for Radar Set Module

```
1    /*
2     * Interface Module:   radar.h
3     * Source:   Generated by 'stetoc' from declaration file: radar.ste
4     * Date Generated:     Mon Aug 25 13:29:13 1997
5     */
6
7    #ifndef _RADAR_H
8    #define _RADAR_H
9
10   #include <types.h>
11   #include <set.h>
12   #include <fsm.h>
13
14   /* Type definition for STE Variables of type RADARVAR */
15
16   typedef struct
17   {
18      int     state;
19      lock_t  lock;
20      FSM     *fsm;
21
22   } RADARVAR;
23
24   #define NILRADAR ((RADARVAR *)0)
```

```
25
26
27  /* Declarations for constants of type RADARVAR */
28
29  extern const RADARVAR *TRKREL;
30  extern const RADARVAR *RNGREL;
31  extern const RADARVAR *STBYREL;
32  extern const RADARVAR *TRKNOT;
33  extern const RADARVAR *RNGNOT;
34  extern const RADARVAR *STBYNOT;
35  extern const RADARVAR *FAILED;
36
37
38  /* Function prototypes: STE Variable Operations */
39
40  extern RADARVAR *CreateRadar( const RADARVAR * );
41  extern int UpdateRadar( RADARVAR *, int );
42  extern int DeleteRadar( RADARVAR * );
43
44
45  /* Function prototypes: Set Inquiry Operations */
46
47  extern Boolean IsRanging( RADARVAR * );
48  extern Boolean IsIsstandby( RADARVAR * );
49  extern Boolean IsIsreliable( RADARVAR * );
50  extern Boolean IsNotreliable( RADARVAR * );
51  extern Boolean IsTracking( RADARVAR * );
52
53
54  /* Function prototypes: Relation Inquiry Operations */
55
56  extern Boolean IsReliable( RADARVAR *, RADARVAR * );
57  extern Boolean IsUnreliable( RADARVAR *, RADARVAR * );
58  extern Boolean IsTrack( RADARVAR *, RADARVAR * );
59  extern Boolean IsRange( RADARVAR *, RADARVAR * );
60  extern Boolean IsStandby( RADARVAR *, RADARVAR * );
61  extern Boolean IsFailure( RADARVAR *, RADARVAR * );
62
63
64  /* Function prototypes: Wait on Transition Event */
65
66  extern int awaitEvtReliable( RADARVAR * );
67  extern int awaitEvtUnreliable( RADARVAR * );
68  extern int awaitEvtTrack( RADARVAR * );
69  extern int awaitEvtRange( RADARVAR * );
70  extern int awaitEvtStandby( RADARVAR * );
```

```
71   extern int awaitEvtFailure( RADARVAR * );
72
73
74   /* Function prototypes: Wait on Condition */
75
76   extern int awaitCndtReliable( RADARVAR *, RADARVAR * );
77   extern int awaitCndfReliable( RADARVAR *, RADARVAR * );
78   extern int awaitCndtUnreliable( RADARVAR *, RADARVAR * );
79   extern int awaitCndfUnreliable( RADARVAR *, RADARVAR * );
80   extern int awaitCndtTrack( RADARVAR *, RADARVAR * );
81   extern int awaitCndfTrack( RADARVAR *, RADARVAR * );
82   extern int awaitCndtRange( RADARVAR *, RADARVAR * );
83   extern int awaitCndfRange( RADARVAR *, RADARVAR * );
84   extern int awaitCndtStandby( RADARVAR *, RADARVAR * );
85   extern int awaitCndfStandby( RADARVAR *, RADARVAR * );
86   extern int awaitCndtFailure( RADARVAR *, RADARVAR * );
87   extern int awaitCndfFailure( RADARVAR *, RADARVAR * );
88
89
90   /* Function prototypes: Wait on Call */
91
92   extern int awaitCallReliable( RADARVAR * );
93   extern int awaitCallUnreliable( RADARVAR * );
94   extern int awaitCallTrack( RADARVAR * );
95   extern int awaitCallRange( RADARVAR * );
96   extern int awaitCallStandby( RADARVAR * );
97   extern int awaitCallFailure( RADARVAR * );
98
99
100  /* Function prototypes: Wait on Set Membership */
101
102  extern int awaitMbrtRanging( RADARVAR * );
103  extern int awaitMbrfRanging( RADARVAR * );
104  extern int awaitMbrtIsstandby( RADARVAR * );
105  extern int awaitMbrfIsstandby( RADARVAR * );
106  extern int awaitMbrtIsreliable( RADARVAR * );
107  extern int awaitMbrfIsreliable( RADARVAR * );
108  extern int awaitMbrtNotreliable( RADARVAR * );
109  extern int awaitMbrfNotreliable( RADARVAR * );
110  extern int awaitMbrtTracking( RADARVAR * );
111  extern int awaitMbrfTracking( RADARVAR * );
112
113
114  /* Function prototypes: Wait on Set Transition */
115
116  extern int awaitSettRanging( RADARVAR * );
```

```
117 extern int awaitSetfRanging( RADARVAR * );
118 extern int awaitSettIsstandby( RADARVAR * );
119 extern int awaitSetfIsstandby( RADARVAR * );
120 extern int awaitSettIsreliable( RADARVAR * );
121 extern int awaitSetfIsreliable( RADARVAR * );
122 extern int awaitSettNotreliable( RADARVAR * );
123 extern int awaitSetfNotreliable( RADARVAR * );
124 extern int awaitSettTracking( RADARVAR * );
125 extern int awaitSetfTracking( RADARVAR * );
126
127
128 /* Function prototypes: State Transition Operations */
129
130 extern int Reliable( RADARVAR * );
131 extern int Unreliable( RADARVAR * );
132 extern int Track( RADARVAR * );
133 extern int Range( RADARVAR * );
134 extern int Standby( RADARVAR * );
135 extern int Failure( RADARVAR * );
136
137
138 #endif
```

C.2 Implementation for Radar Set Module

```
1    /*
2     * Implementation Module: radar.c
3     * Source:        Generated by 'stetoc' from declaration file: radar.ste
4     * Date Generated:          Mon Aug 25 13:29:13 1997
5     */
6
7    #include <stdio.h>
8    #include <stdlib.h>
9    #include <set.h>
10   #include "radar.h"
11
12   /* Some Constants */
13
14   #define NUMSTATES      7
15   #define NUMRELATIONS   6
16   #define NUMSETS        5
17
18   /* Constants for type RADARVAR */
19
20   static const RADARVAR TRKREL_CONSTANT = { 1, 0, NILFSM };
21   const RADARVAR *TRKREL = &TRKREL_CONSTANT;
22
23   static const RADARVAR RNGREL_CONSTANT = { 2, 0, NILFSM };
24   const RADARVAR *RNGREL = &RNGREL_CONSTANT;
25
26
27
28   /* Set Data for Ranging */
29
30   static uint RangingBitString[] =
31   {
32      0x0012
33   };
34
35   static SET RangingSet =
36   {
37      2, 1, RangingBitString
38   };
39
40
41
42
43   /* Set Table */
44
```

```
45  static SETTABLEENTRY SetTable[] =
46  {
47     &RangingSet, False,
48     &IsstandbySet, False,
49     &IsreliableSet, False,
50     &NotreliableSet, False,
51     &TrackingSet, False
52  },
53
54  /* Handles for Sets */
55
56  #define RANGINGSET 0
57  #define ISSTANDBYSET 1
58  #define ISRELIABLESET 2
59  #define NOTRELIABLESET 3
60  #define TRACKINGSET 4
61
62
63  /* Transition Relation */
64
65  static int Radarx[NUMSTATES][NUMRELATIONS] =
66  {
67     -1,  4, -1,  2,  3,  7,
68     -1,  5,  1, -1,  3,  7,
69     -1,  6,  1,  2, -1,  7,
70      1, -1, -1,  5,  6,  7,
71      2, -1,  4, -1,  6,  7,
72      3, -1,  4,  5, -1,  7,
73     -1, -1, -1, -1, -1, -1
74  },
75
76  static int *Radar[NUMSTATES] =
77  {
78     Radarx[0], Radarx[1], Radarx[2], Radarx[3], Radarx[4], Radarx[5], Radarx[6]
79  },
80
81  /* Handles for Relations */
82
83  #define RELIABLERELATION 0
84  #define UNRELIABLERELATION 1
85  #define TRACKRELATION 2
86  #define RANGERELATION 3
87  #define STANDBYRELATION 4
88  #define FAILURERELATION 5
89
90
```

```
91   /* Condition Table */
92
93   static CONDTABLEENTRY CondTable[] =
94   {
95          RELIABLERELATION,  2,  1, False,
96          RELIABLERELATION,  1,  2, False,
97          RELIABLERELATION,  3,  3, False,
98
99        UNRELIABLERELATION,  5,  4, False,
100       UNRELIABLERELATION,  4,  5, False,
101       UNRELIABLERELATION,  6,  6, False,
102
103            TRACKRELATION,  4,  7, False,
104            TRACKRELATION,  1,  8, False,
105
106            RANGERELATION,  5,  9, False,
107            RANGERELATION,  2, 10, False,
108
109          STANDBYRELATION,  6, 11, False,
110          STANDBYRELATION,  3, 12, False,
111
112          FAILURERELATION,  7, 13, False,
113
114  };
115
116
117  /* STE Variable Operations */
118
119  RADARVAR *CreateRadar( const RADARVAR *p )
120  {
121     int CurState = p->state;
122     RADARVAR *ste;
123
124     if ((ste = (RADARVAR *) malloc(sizeof(RADARVAR))) == NULL)
125        return( NILRADAR );
126
127     ste->state = CurState;
128     ste->lock = 0;
129
130     if ((ste->fsm = CreateMachine( NUMSETS, NUMSTATES, NUMRELATIONS,
131                        NUMCONDITIONS, CurState, SetTable,
132                        Radar, CondTable )) == NILFSM )
133     {
134        free( ste );
135        return( NILRADAR );
136     }
```

```
137
138     return( ste );
139  }
140
141  int UpdateRadar( RADARVAR *p, int relation )
142  {
143     return( UpdateMachine((STEOBJECT *)p, relation) );
144  }
145
146  int DeleteRadar( RADARVAR *p )
147  {
148     int restarted = DeleteMachine((STEOBJECT *)p );
149     free( p );
150     return ( restarted );
151  }
152
153
154  /* Set Inquiry Operations */
155
156  Boolean IsRanging( RADARVAR *p )
157  {
158     SETTABLE *SetTable = p->fsm->SetTable;
159     return( IsInSet( SetTable[RANGINGSET].set, p->state ));
160  }
161
162
163
164
165  /* Relation Inquiry Operations */
166
167  Boolean IsReliable( RADARVAR *p1, RADARVAR *p2 )
168  {
169     int lc = p1->state;
170     int rc = p2->state;
171     int **tm = p1->fsm->Transition;
172     return( (tm[lc-1][RELIABLERELATION] == rc) ? True : False );
173  }
174
175
176
177
178  /* Synchronization Operators: Wait on Transition Event */
179
180  int awaitEvtReliable( RADARVAR *p )
181  {
182     SEMAPHORE sema = p->fsm->EvtSema[RELIABLERELATION];
```

```
183    return( dop_pass( sema ));
184 }
185
186 int awaitEvtTrack( RADARVAR *p )
187 {
188    SEMAPHORE sema = p->fsm->EvtSema[TRACKRELATION];
189    return( dop_pass( sema ));
190 }
191
192
193
194 /* Synchronization Operators: Wait on Condition */
195
196 int awaitCndtTrack( RADARVAR *p1, RADARVAR *p2 )
197 {
198    SEMAPHORE sema;
199    int       handle;
200
201    if ((handle = CondMatrix[TRACKRELATION][(p2->state)-1]) <= 0)
202       return( -1 );
203    sema = p1->fsm->CndtSema[handle-1];
204    return( dop_pass( sema ));
205 }
206
207 int awaitCndfTrack( RADARVAR *p1, RADARVAR *p2 )
208 {
209    SEMAPHORE sema;
210    int       handle;
211
212    if ((handle = CondMatrix[TRACKRELATION][(p2->state)-1]) <= 0)
213       return( -1 );
214    sema = p1->fsm->CndfSema[handle-1];
215    return( dop_pass( sema ));
216 }
217
218
219 /* Synchronization Operators: Wait on Call */
220
221 int awaitCallTrack( RADARVAR *p )
222 {
223    SEMAPHORE  sema = p->fsm->CallSema[TRACKRELATION];
224    return( dop_pass( sema ));
225 }
226
227 int awaitCallRange( RADARVAR *p )
228 {
```

```
229      SEMAPHORE  sema = p->fsm->CallSema[RANGERELATION];
230      return( dop_pass( sema ));
231   }
232
233
234   /* Synchronization Operators: Wait on Set Membership */
235
236   int awaitMbrtRanging( RADARVAR *p )
237   {
238      SEMAPHORE tsema = p->fsm->MbrtSema[RANGINGSET];
239      return( dop_pass( tsema ));
240   }
241
242   int awaitMbrfRanging( RADARVAR *p )
243   {
244      SEMAPHORE fsema = p->fsm->MbrfSema[RANGINGSET];
245      return( dop_pass( fsema ));
246   }
247
248
249   /* Synchronization Operators: Wait on Set Transition */
250
251   int awaitSettRanging( RADARVAR *p )
252   {
253      SEMAPHORE ssema = p->fsm->SSettSema[RANGINGSET];
254      SEMAPHORE lsema = p->fsm->LSettSema[RANGINGSET];
255
256      if ( IsRanging( p ) == True )
257         return( dop_pass( lsema ));
258      else
259         return( dop_pass( ssema ));
260   }
261
262   int awaitSetfRanging( RADARVAR *p )
263   {
264      SEMAPHORE ssema = p->fsm->SSetfSema[RANGINGSET];
265      SEMAPHORE lsema = p->fsm->LSetfSema[RANGINGSET];
266
267      if ( IsRanging( p ) == True )
268         return( dop_pass( ssema ));
269      else
270         return( dop_pass( lsema ));
271   }
272
273
274   /* State Transition Operations */
```

```
275
276 int Reliable( RADARVAR *p )
277 {
278    return( UpdateRadar( p, RELIABLERELATION )),
279 }
280
281 int Range( RADARVAR *p )
282 {
283    return( UpdateRadar( p, RANGERELATION )),
284 }
285
286
287
```

Appendix D

Programming Interfaces

D.1 STEM Operations

Declaration `#include<type.h>`

Description In this section *Type* indicates the name of a STE declaration, *Set* represents a *set attribute*, *Relation* represents a *relation attribute* and p, q are STE variables. A STE declaration is translated into an implementation using the `stetoc` precompiler. Every implementation has an associated interface which is placed in file *type.h*.

Structures

```
typedef struct
{
    int      state;    // current state of STEM
    lock_t   lock;     // spinlock for mutual exclusion
    FSM      *fsm;     // the STEM
} TYPEVAR;
```

Members

Create*Type*

```
TYPEVAR *CreateType( const TYPEVAR *p );
```

`CreateType()` is the variable constructor for STE variables belonging to class *Type*. The constructor uses the current state of a reference object, p, and transition manager function `CreateMachine()` to initialize the STE variable.

Return values. If successful, this function returns a pointer to an object of type *TYPEVAR*, otherwise, NULL will be returned.

Delete*Type*

`int Delete`*`Type`*`( `*`TYPEVAR`*` *p );`

Delete*Type*() is the variable destructor for STE variables belonging to class *Type*. This function passes control directly to the transition manager function DeleteMachine(). The following housekeeping functions are performed by DeleteMachine().

- the dop_up() operation is executed on every semaphore referenced in p. This forces all blocked threads to restart.
- the dop_delete() operation is executed on every semaphore referenced in p. This recovers all kernel resource associated with the semaphore.
- the memory allocated to the semaphore arrays and tables is returned to the heap.

Return values. The value returned by this operation indicates the number of threads restarted by the destructor.

Update*Type*

`int Update`*`Type`*`( `*`TYPEVAR`*` *p, int event );`

Update*Type*() is the variable update operation for STE variables belonging to class *Type*. This function passes control directly to the transition manager function UpdateMachine(). The following functions are performed by UpdateMachine().

- uses event to determine the next state of the STE state machine.
- uses the next state information to update the set and condition tables.
- uses the information in the set and condition tables to update all semaphores.

Return values. Returns 0 if the update was successful, otherwise, -1 indicates that the function was unable to compute the next state of the STE state machine.

Is*Set*

`Boolean Is`*`Set`*`( `*`TYPEVAR`*` *p );`

Is*Set*() returns True if the current state of STE variable p belongs to set *Set*. It returns False otherwise.

Is*Relation* `Boolean IsRelation( TYPEVAR *p, TYPEVAR *q );`

`IsRelation()` returns `True` if the relation ***Relation*** contains the transition `(p,q)` ∈ ***Relation***. It returns `False` otherwise.

Relation `int Relation( TYPEVAR *p );`

Relation`()` is a state transition operation for STE variables belonging to class ***Type***. There is one transition operation for each *relation attribute* in the declaration for class ***Type***. This function passes control directly to the update operation `Update`***Type***`()`.

Return values: Returns 0 if the transition operation was successful, otherwise, -1 indicates that the update operation was unable to compute the next state of the STE state machine.

Synchronization Operators

The synchronization operations are built upon the d-operations. In fact the only external function used by the synchronization operations is function `dop_pass()`.

Return values: Returns 0 if the thread passed the semaphore without blocking, 1 if the thread blocks then is subsequently interrupted by an event it should respond to, and, 2 if the thread blocks then is subsequently restarted by to `dop_up()` operation. A return value of -1 indicates a system error. At present the only system error returned by the `dop_pass()` operation is `ESRCH` which indicates that the associated semaphore handle is invalid.

awaitEvt*Relation* `int awaitEvtRelation( TYPEVAR *p );`

`awaitEvt`***Relation*** is a class of synchronization operation used when a thread should wait for a transition in the specified relation.

awaitCall*Relation* `int awaitCallRelation( TYPEVAR *p );`

`awaitCall`***Relation*** is a class of synchronization operation used when a thread should block until the corresponding state transition operation is called.

awaitSet*Set*

```
int awaitSettSet( TYPEVAR *p );
int awaitSetfSet( TYPEVAR *p );
```

`awaitSetSet` is a class of synchronization operation used when a thread should block under the following circumstances

- wait until the state of a STE variable changes to an element *inside* the set (`awaitSettSet()`)
- wait until the state of a STE variable changes to an element *outside* the set (`awaitSetfSet()`)

awaitCnd*Relation*

```
int awaitCndtRelation( TYPEVAR *p, TYPEVAR *q ),
int awaitCndfRelation( TYPEVAR *p, TYPEVAR *q ),
```

`awaitCndRelation` is a class of synchronization operation used to choose between blocking (`awaitCndtRelation()`) and not blocking (`awaitCndfRelation()`) an executing thread depending on the value of a condition in the condition table

awaitMbr*Set*

```
int awaitMbrtSet( TYPEVAR *p ),
int awaitMbrfSet( TYPEVAR *p ),
```

`awaitMbrSet` is a class of synchronization operation used to choose between blocking (`awaitMbrtSet()`) and not blocking (`awaitMbrfSet()`) an executing thread depending on the value of a condition in the set table

D.2 D-Operations

Declaration

```
#include<kernel h>
```

Description

The d-operations provide the low-level synchronization interface for the STE synchronization operations The semaphores are implemented by the operating system kernel User-processes gain access to these semaphores by requesting handles with the `dop_create()` system call Semaphores can only be manipulated through semaphore handles

Structures

```
typedef long sid_t,
typedef sid_t SEMAPHORE,
```

Members

dop_create

```
sid_t dop_create( int init ),
```

dop_create() is used to create and intialize a STE semaphore. The paramter init can have one of two values. If the initialization value is S_OPENED the semaphore will be initialized in a non-blocking state. If the initialization value is S_CLOSED the semaphore will be initialized in a blocking state.

Return values: A unique *semaphore identifier*. This handle should be a value between 1 and 65535. A return value of -1 indicates a system error. For dop_create() the extended system error will be ENOMEM which indicates the kernel was unable to allocate the resources necessary to create a semaphore.

dop_destroy

```
int dop_destroy( sid_t sema );
```

dop_destroy() is used to surrender a STE semaphore and return its resources to the operating system. This operations input parameter must identify a STE semaphore previously acquired with the dop_create() operation.

Return values: Returns 0 if the delete operation was successful on sema. A return value of -1 indicates a system error. For dop_delete() the extended system errors will be either ESRCH which indicates that sema is an invalid handle, or, EEXIST which indicates that there are still threads blocked on this semaphore.

dop_pass

```
int dop_pass( sid_t sema );
```

dop_pass() is the semaphore passage operation. The semaphore passage operation allows a thread to suspend execution. This occurs if the semaphore count is negative when the function is invoked. Threads suspended on a semaphore will be restarted by the next semaphore opening operation on the same semaphore.

Return values: Returns 0 if the thread passed the semaphore without blocking, 1 if the thread blocks then is subsequently interrupted by an event it should respond to, and, 2 if the thread blocks then is subsequently restarted by the dop_up() operation. A return value of -1 indicates a system error. For dop_pass() the extended system error will be ESRCH which indicates that sema is an invalid handle.

dop_down

```
int dop_down( sid_t sema );
```

dop_down() is the semaphore closing operation. This operation decrements the value of the indicated semaphore. The executing

thread is not blocked. However, threads which subsequently use the semaphore passage operation will be suspended.

Return values: Returns 0 if the down operation was successful on `sema`. A return value of -1 indicates a system error. For `dop_down()` the extended system error will be `ESRCH` which indicates that `sema` is an invalid handle.

dop_up

```
int dop_up( sid_t sema );
```

`dop_up()` is the semaphore opening operation. The opening operation increments the value of the semaphore. If the value of the semaphore becomes nonnegative, all threads suspended on the semaphore are restarted.

Return values: The number of threads restarted. A return value of -1 indicates a system error. For `dop_up()` the extended system error will be `ESRCH` which indicates that `sema` is an invalid handle.

dop_updown

```
int dop_updown( sid_t sema );
```

`dop_updown()` combines the effect of `dop_up()` and `dop_down()` in an atomic operation.

Return values: The number of threads restarted. A return value of -1 indicates a system error. For `dop_updown()` the extended system error will be `ESRCH` which indicates that `sema` is an invalid handle.

dop_movesema

```
int dop_movesema( sid_t src, sid_t dst );
```

`dop_movesema()` is the semaphore morphing operation. It is used to move the list of waiting threads on a source semaphore to the waiting list at a destination semaphore.

Return values: The number of threads moved from the source (`src`) to destination (`dst`) semaphore. A return value of -1 indicates a system error. For `dop_movesema()` the extended system error will be `ESRCH` which indicates that either `src` or `dst` is an invalid handle.

D.3 Transition Manager Operations

Declaration `#include<fsm h>`

Description The transition manager contains the logic necessary to create, update and delete any state transition event machine produced by the **stetoc** precompiler (see Section 3 2 on page 31) The transition manager accesses objects of type STEOBJECT All STEMs produced by the **stetoc** precompiler are derived from this data type

Structures

```
typedef struct
{
   int     state,    // current state of STEM
   lock_t  lock,     // spinlock for mutual exclusion
   FSM     *fsm,     // the STEM
} STEOBJECT,
```

Members

CreateMachine

```
FSM *CreateMachine( int numsets, int numstates,
    int numrelations, int numconditions, int curstate,
    SETTABLE *setTable, int **trans,
    CONDTABLE *condTable ),
```

CreateMachine() is a generic constructor used to create the STE state machine member of STE variables The parameters **numsets** (number of sets), **numstates** (number of states), **numrelations** (number of relations), and, **numconditions** (number of conditions) are used to dimension the semaphore arrays created by this operation Copies of **setTable** (set table) and **condTable** (condition table) are made for each STE variable It then performs the following initializations

- initializes all entries in the set table and condition table
- creates and initializes the semaphore arrays required for each synchronization operation

Return values If successful, this function returns a pointer to an instance of the STE state machine, otherwise, NULL will be returned

DeleteMachine

```
int DeleteMachine( STEOBJECT *obj ),
```

DeleteMachine() is used to recover all of the resources associated with the STE state machine component of obj The housekeeping functions performed by DeleteMachine() include

- applying the dop_up() operation to every semaphore referenced in obj This forces all blocked threads to restart
- applying the dop_delete() operation to every semaphore referenced in obj This recovers all kernel resource associated with the semaphore
- the memory allocated to the semaphore arrays and tables is returned to the heap

Return values The value returned by this operation indicates the number of threads that were restarted during destruction

UpdateMachine `int UpdateMachine( STEOBJECT *obj, int event ),`

UpdateMachine() is a generic update operation able to update any class of STE variable derived from the STEOBJECT data type To perform the update this operation requires a reference to the object affected (obj) and the event generating the update The following functions are performed by UpdateMachine()

- uses event to determine the next state of the STE state machine
- uses the next state information to update the set and condition tables
- uses the information in the set and condition tables to update all semaphores

Return values Returns 0 if the update was successful, otherwise, -1 indicates that the function was unable to compute the next state of the STE state machine

D.4 Set Operations

Declaration `#include<set h>`

Description The set module implements a *set of integers* abstract data type This module is used primarily by the precompiler to implement the STEM's *set table* and the set inquiry operations

Structures

```
typedef unsigned int uint;

typedef struct
{
   uint nmembers;       // No. of members
   uint naunits;        // No. of allocation units
   uint *BitString;     // A pointer to storage
} SET;
```

Members

CreateSet

```
SET *CreateSet( uint size );
```

`CreateSet()` is used to create a set of integers. The set will is initially empty.

Return values: If initialization is successful, this function returns a pointer to a set of integers able to hold `size` members. Otherwise `NULL` will be returned.

DisposeSet

```
void DisposeSet( SET ** set );
```

`DisposeSet()` is used to delete a set of integers previously created by the `CreateSet()` operation.

Return values: The pointer referenced by `set` is reset to `NULL`.

AddToSet

```
Boolean AddToSet( SET * set, uint member );
```

`AddToSet()` adds the integer `member` to `set`.

Return values: Return `True` if `member` was added to `set`, otherwise, return `False`.

RemoveFromSet

```
void RemoveFromSet( SET * set, uint member );
```

`RemoveFromSet()` removes the integer `member` from `set`.

IsInSet

```
Boolean IsInSet( SET * set, uint member );
```

`IsInset()` tests wheter `member` is an element of `set`.

Return values: Return `True` if `member` is a member of `set`, otherwise, return `False`.

Appendix E

Overview of Tool Set

This chapter discusses the usage of two tools developed for working with STE type declarations. `stetoc` is used to generate compilable 'C' code from a STE type declaration. `verfiy` is used to test the operation of the Radar Set Module generated by `stetoc`.

E.1 stetoc - A STE to 'C' precompiler

In Section 1.2 we have demonstrated how to create a STE type declaration from a formal specification of a system. In this section we describe how `stetoc` can be used to convert such a declaration to a compilable implementation.

Usage: `stetoc [-p | -t | -g] -o base stefile`

Options:

```
-p          Pretty print STE declaration file
-t          Type check STE declaration file
-g          Generate STE operations
-o base     Use base as basename of output files
stefile     STE declaration file
```

We have followed a convention that files with the '* ste' file extension contain STE type declarations With the -g option, stetoc will generate two files associated with the STE declaration base h an interface module and base c an implementation module base is generally the base name of the STE declaration file (stefile) This can be overriden using the -o option Options -p and -t provide no compilable output

As an example, we converted the STE type declaration for the Radar Set Module given in Appendix A 1 using the following command line stetoc -g radar ste The implementation produced by stetoc for the Radar Set Module is shown in Appendices C 1 and C 2 More details on the operation of stetoc can be found in Section 3 2

E.2 verify - Testing Radar Set Module Operations

verify and the awk script verify awk have been used to test the operation of the Radar Set Module This is achieved by creating a simulation of the Radar Set Module (described in more detail in Section 5 1) and injecting a sequence of events into the simulation A number of test processes respond to the input of these events and journal their response to a simulation log After the simulation completes, the simulation log can be analyzed for errors using the awk script We use these two programs as follows

Usage

```
verify -f tracefile
awk -f verify awk logfile >resfile
```

Options:

```
-f tracefile    Input event trace for Radar Set Module
```

We have followed the conventions that files with the '`*.trc`' file extension contain input event traces and files with the '`*.log`' file extension contain simulation logs. **verify** generates a simulation log called **base.log** where **base** is the base name of the trace file. The awk script analyzes this log file and outputs the result to the standard output.

As an example, examine the output resulting from the following command sequence:

```
verify -f canon.trc
awk -f canon.log >canon.res
```

The trace file **canon.trc** consists of the following line:

```
arbatbasbatrsbrtstrsaq
```

This input file describes the sequence of events that will take place during the simulation. The trace is nothing more than a sequence of input characters coded for specific events. For this trace the events were coded for the Radar Set Module: r = range, s = standby, t = track, a = reliable, b = unreliable and q = quit.

The simulation log for this trace summarizes all significant simulation events. An extract from the simulation log for the above trace is shown below:

```
471649    328    192    mm-start slice=7 clock=471702
473947    329    192    mm(Range) TRKREL->RNGREL restarts=18
474360    330    192    mm-end slice=8 clock=474429
474782    331    204    restart-0-awaitEvtRange 2
475080    332    204    pass-1-awaitEvtRange
475412    333    205    restart-0-awaitEvtRange 2
475716    334    205    pass-1-awaitEvtRange
476038    335    206    restart-0-awaitEvtRange 2
476334    336    206    pass-1-awaitEvtRange
```

The first column contains the time on the simulation clock. This is followed by a record number and process identifier. The last column contains the process response. For example, record 329 logs the occurrence of a `range` event. A more detailed log for this trace can be found in Appendix F.1.

When the simulation log generated by `verify` for the above trace is analyzed, we obtain the following results:

```
1     Range: 4, Track: 4, Standby: 4, Reliable: 5, Unreliable: 4
2     Events processed: 21, Operations checked: 303
3     Assertions tripped: 0, Verification failures: 0
4
5     Intervals between events (ms):
6
7        16.3     40.2     60.1     19.9     40.0
8        60.0     20.1     39.9     59.9     20.1
9        59.9     60.0     40.1     20.0     59.9
10       60.0     60.0     60.1     60.0     40.1
11
```

For this test run no errors were reported by the analyzer.

Appendix F

State Machine Verification

F.1 Example Simulation Log

```
1        Simulation  Record   Thread          Thread
2          Clock     Number  Identifier      Response
3    ----------------------------------------------------------------------
4            47        1        192       mm-begin clock=49
5          6486        2        192       mm-end slice=0 clock=6518
6         31789        3        193       rm-begin clock=36
7         37481        4        193       rm(Reliable) STBYNOT->STBYREL restarts=0
8         37927        5        193       rm-end slice=0 clock=6188
9         51689        6        192       mm-start slice=0 clock=51744
10        53815        7        192       mm(Range) STBYREL->RNGREL restarts=0
11        54227        8        192       mm-end slice=1 clock=54295
12        71784        9        194       fm-begin clock=33
13        81243       10        195       pass-0-awaitEvtReliable
14        81773       11        196       pass-0-awaitEvtReliable
15        82143       12        197       pass-0-awaitEvtReliable
16        87498       13        194       fm-end slice=0 clock=15725
17        91660       14        193       rm-start slice=0 clock=59902
18        94052       15        193       rm(Unreliable) RNGREL->RNGNOT restarts=0
19        94470       16        193       rm-end slice=1 clock=62732
20        95593       17        198       pass-0-awaitEvtUnreliable
21        96106       18        199       pass-0-awaitEvtUnreliable
22        96560       19        200       pass-0-awaitEvtUnreliable
23        97021       20        201       pass-0-awaitEvtTrack
24        97494       21        202       pass-0-awaitEvtTrack
```

25	97947	22	203	pass-0-awaitEvtTrack
26	98382	23	204	pass-0-awaitEvtRange
27	98822	24	205	pass-0-awaitEvtRange
28	99265	25	206	pass-0-awaitEvtRange
29	99710	26	207	pass-0-awaitEvtStandby
30	100147	27	208	pass-0-awaitEvtStandby
31	100611	28	209	pass-0-awaitEvtStandby
32	101026	29	210	pass-T-0-awaitCndfReliable
33	101448	30	211	pass-T-0-awaitCndfReliable
34	101855	31	212	pass-T-0-awaitCndfReliable
35	102269	32	213	pass-F-0-awaitCndtUnreliable
36	102684	33	214	pass-F-0-awaitCndtUnreliable
37	111822	34	192	mm-start slice=1 clock=111876
38	113400	35	192	mm-end slice=2 clock=113450
39	113883	36	215	pass-F-0-awaitCndtUnreliable
40	114312	37	216	pass-F-0-awaitCndtTrack
41	114715	38	217	pass-F-0-awaitCndtTrack
42	115105	39	218	pass-F-0-awaitCndtTrack
43	115507	40	219	pass-F-0-awaitCndtRange
44	115911	41	220	pass-F-0-awaitCndtRange
45	116305	42	221	pass-F-0-awaitCndtRange
46	116716	43	222	pass-F-0-awaitCndtStandby
47	117124	44	223	pass-F-0-awaitCndtStandby
48	117522	45	224	pass-F-0-awaitCndtStandby
49	117936	46	225	pass-T-0-awaitMbrfRanging
50	118351	47	226	pass-T-0-awaitMbrfRanging
51	118743	48	227	pass-T-0-awaitMbrfRanging
52	119151	49	228	pass-F-0-awaitMbrtIsstandby
53	119562	50	229	pass-F-0-awaitMbrtIsstandby
54	119969	51	230	pass-F-0-awaitMbrtIsstandby
55	120391	52	231	pass-F-0-(F->T)-awaitSettTracking
56	120820	53	232	pass-F-0-(F->T)-awaitSettTracking
57	123153	54	233	pass-F-0-(F->T)-awaitSettTracking
58	123663	55	234	pass-F-0-(F->T)-awaitSettIsreliable
59	124091	56	235	pass-F-0-(F->T)-awaitSettIsreliable
60	124499	57	236	pass-F-0-(F->T)-awaitSettIsreliable
61	131724	58	194	fm-start slice=0 clock=59967
62	133322	59	194	fm-end slice=1 clock=61566
63	151675	60	193	rm-start slice=1 clock=119924
64	154137	61	193	rm(Reliable) RNGNOT->RNGREL restarts=15
65	154557	62	193	rm-end slice=2 clock=122818
66	154986	63	195	restart-0-awaitEvtReliable 2
67	155292	64	195	pass-1-awaitEvtReliable
68	155633	65	196	restart-0-awaitEvtReliable 2
69	155934	66	196	pass-1-awaitEvtReliable
70	156255	67	197	restart-0-awaitEvtReliable 2

```
71    156554   68    197   pass-1-awaitEvtReliable
72    156883   69    210   restart-T-0-awaitCndfReliable 2
73    157201   70    210   pass-F-1-awaitCndtReliable
74    157544   71    211   restart-T-0-awaitCndfReliable 2
75    157850   72    211   pass-F-1-awaitCndtReliable
76    158189   73    212   restart-T-0-awaitCndfReliable 2
77    158499   74    212   pass-F-1-awaitCndtReliable
78    158847   75    216   restart-F-0-awaitCndtTrack 2
79    159158   76    216   pass-T-1-awaitCndfTrack
80    159498   77    217   restart-F-0-awaitCndtTrack 2
81    159795   78    217   pass-T-1-awaitCndfTrack
82    160121   79    218   restart-F-0-awaitCndtTrack 2
83    160429   80    218   pass-T-1-awaitCndfTrack
84    160763   81    222   restart-F-0-awaitCndtStandby 2
85    161058   82    222   pass-T-1-awaitCndfStandby
86    161393   83    223   restart-F-0-awaitCndtStandby 2
87    161699   84    223   pass-T-1-awaitCndfStandby
88    162037   85    224   restart-F-0-awaitCndtStandby 2
89    162336   86    224   pass-T-1-awaitCndfStandby
90    162674   87    234   restart-F-0-(F->T)-awaitSettIsreliable 2
91    163504   88    234   pass-T-1-(T->F->T)-awaitSettIsreliable
92    163934   89    235   restart-F-0-(F->T)-awaitSettIsreliable 2
93    164267   90    235   pass-T-1-(T->F->T)-awaitSettIsreliable
94    164597   91    236   restart-F-0-(F->T)-awaitSettIsreliable 2
95    164904   92    236   pass-T-1-(T->F->T)-awaitSettIsreliable
96
97                       ... (records 93-327)
98
99    471649   328   192   mm-start slice=7 clock=471702
100   473947   329   192   mm(Range) TRKREL->RNGREL restarts=18
101   474360   330   192   mm-end slice=8 clock=474429
102   474782   331   204   restart-0-awaitEvtRange 2
103   475080   332   204   pass-1-awaitEvtRange
104   475412   333   205   restart-0-awaitEvtRange 2
105   475716   334   205   pass-1-awaitEvtRange
106   476038   335   206   restart-0-awaitEvtRange 2
107   476334   336   206   pass-1-awaitEvtRange
108   476652   337   213   restart-T-5-awaitCndfUnreliable 2
109   476964   338   213   pass-F-6-awaitCndtUnreliable
110   477304   339   214   restart-T-5-awaitCndfUnreliable 2
111   477606   340   214   pass-F-6-awaitCndtUnreliable
112   477952   341   215   restart-T-5-awaitCndfUnreliable 2
113   478260   342   215   pass-F-6-awaitCndtUnreliable
114   478607   343   216   restart-F-6-awaitCndtTrack 2
115   478892   344   216   pass-T-7-awaitCndfTrack
116   479223   345   217   restart-F-6-awaitCndtTrack 2
```

```
117    479516   346   217    pass-T-7-awaitCndfTrack
118    479840   347   218    restart-F-6-awaitCndtTrack 2
119    480150   348   218    pass-T-7-awaitCndfTrack
120    480497   349   228    restart-F-1-awaitMbrtRanging 2
121    480816   350   228    pass-F-2-awaitMbrtIsstandby
122    481161   351   229    restart-F-1-awaitMbrtRanging 2
123    481479   352   229    pass-F-2-awaitMbrtIsstandby
124    481828   353   230    restart-F-1-awaitMbrtRanging 2
125    482145   354   230    pass-F-2-awaitMbrtIsstandby
126    482485   355   225    restart-T-3-awaitMbrfTracking 2
127    482773   356   225    pass-T-4-awaitMbrfRanging
128    483117   357   226    restart-T-3-awaitMbrfTracking 2
129    483826   358   226    pass-T-4-awaitMbrfRanging
130    484200   359   227    restart-T-3-awaitMbrfTracking 2
131    484495   360   227    pass-T-4-awaitMbrfRanging
132    484833   361   231    restart-T-2-(T->F)-awaitSetfTracking 2
133    485180   362   231    pass-F-3-(F->T->F)-awaitSetfTracking
134    485545   363   232    restart-T-2-(T->F)-awaitSetfTracking 2
135    485850   364   232    pass-F-3-(F->T->F)-awaitSetfTracking
136    486216   365   233    restart-T-2-(T->F)-awaitSetfTracking 2
137    486554   366   233    pass-F-3-(F->T->F)-awaitSetfTracking
138
139                              (records 367-622)
140
141    931666   623   193    rm-start slice=14 clock=899916
142    933998   624   193    rm(Reliable) STBYNOT->STBYREL restarts=12
143    934421   625   193    rm-end slice=15 clock=902683
144    934847   626   195    restart-3-awaitEvtReliable 2
145    935155   627   195    pass-4-awaitEvtReliable
146    935499   628   196    restart-3-awaitEvtReliable 2
147    935800   629   196    pass-4-awaitEvtReliable
148    936119   630   197    restart-3-awaitEvtReliable 2
149    936413   631   197    pass-4-awaitEvtReliable
150    936744   632   216    restart-F-8-awaitCndtTrack 2
151    937056   633   216    pass-T-9-awaitCndfTrack
152    937390   634   217    restart-F-8-awaitCndtTrack 2
153    937687   635   217    pass-T-9-awaitCndfTrack
154    938014   636   218    restart-F-8-awaitCndtTrack 2
155    938320   637   218    pass-T-9-awaitCndfTrack
156    938640   638   219    restart-T-9-awaitCndfRange 2
157    938931   639   219    pass-F-10-awaitCndtRange
158    939258   640   220    restart-T-9-awaitCndfRange 2
159    939543   641   220    pass-F-10-awaitCndtRange
160    939884   642   221    restart-T-9-awaitCndfRange 2
161    940194   643   221    pass-F-10-awaitCndtRange
162    940542   644   234    restart-F-4-(F->T)-awaitSettIsreliable 2
```

163	940892	645	234	pass-T-5-(T->F->T)-awaitSettIsreliable
164	941651	646	235	restart-F-4-(F->T)-awaitSettIsreliable 2
165	942010	647	235	pass-T-5-(T->F->T)-awaitSettIsreliable
166	942349	648	236	restart-F-4-(F->T)-awaitSettIsreliable 2
167	942658	649	236	pass-T-5-(T->F->T)-awaitSettIsreliable
168	951685	650	192	mm-start slice=15 clock=951743
169	953551	651	192	mm-exit clock=953603
170	971828	652	194	fm-start slice=14 clock=900054
171	972845	653	194	fm-exit clock=901090
172	991804	654	193	rm-start slice=15 clock=960031
173	992896	655	193	rm-exit clock=961146
174	1013335	656	195	restart-4-awaitEvtReliable 2
175	1017656	657	196	restart-4-awaitEvtReliable 2
176	1021917	658	197	restart-4-awaitEvtReliable 2
177	1026231	659	198	restart-3-awaitEvtUnreliable 2
178	1030489	660	199	restart-3-awaitEvtUnreliable 2
179	1034841	661	200	restart-3-awaitEvtUnreliable 2
180	1039109	662	201	restart-4-awaitEvtTrack 2
181	1043406	663	202	restart-4-awaitEvtTrack 2
182	1047684	664	203	restart-4-awaitEvtTrack 2
183	1052020	665	204	restart-3-awaitEvtRange 2
184	1056362	666	205	restart-3-awaitEvtRange 2
185	1060628	667	206	restart-3-awaitEvtRange 2
186	1064890	668	207	restart-4-awaitEvtStandby 2
187	1069146	669	208	restart-4-awaitEvtStandby 2
188	1073588	670	209	restart-4-awaitEvtStandby 2
189	1077893	671	210	restart-F-5-awaitCndtReliable 2
190	1082180	672	211	restart-F-5-awaitCndtReliable 2
191	1086485	673	212	restart-F-5-awaitCndtReliable 2
192	1090812	674	213	restart-F-6-awaitCndtUnreliable 2
193	1095256	675	214	restart-F-6-awaitCndtUnreliable 2
194	1099579	676	215	restart-F-6-awaitCndtUnreliable 2
195	1103919	677	216	restart-T-9-awaitCndfTrack 2
196	1108261	678	217	restart-T-9-awaitCndfTrack 2
197	1112670	679	218	restart-T-9-awaitCndfTrack 2
198	1117020	680	219	restart-F-10-awaitCndtRange 2
199	1121384	681	220	restart-F-10-awaitCndtRange 2
200	1125744	682	221	restart-F-10-awaitCndtRange 2
201	1130099	683	222	restart-F-6-awaitCndtStandby 2
202	1134668	684	223	restart-F-6-awaitCndtStandby 2
203	1139096	685	224	restart-F-6-awaitCndtStandby 2
204	1143485	686	228	restart-F-7-awaitMbrtRanging 2
205	1147890	687	229	restart-F-7-awaitMbrtRanging 2
206	1152390	688	230	restart-F-7-awaitMbrtRanging 2

F.2 Verification Results

F.2.1 Verification results for canon

```
1    Range: 4, Track: 4, Standby: 4, Reliable: 5, Unreliable: 4
2    Events processed: 21, Operations checked: 303
3    Assertions tripped: 0, Verification failures: 0
4
5    Intervals between events (ms):
6
7       16.3     40.2     60.1     19.9     40.0
8       60.0     20.1     39.9     59.9     20.1
9       59.9     60.0     40.1     20.0     59.9
10      60.0     60.0     60.1     60.0     40.1
11
```

F.2.2 Verification results for canon20

```
1    Range: 40, Track: 40, Standby: 40, Reliable: 40, Unreliable: 40
2    Events processed: 200, Operations checked: 3203
3    Assertions tripped: 0, Verification failures: 0
4
5    Intervals between events (ms):
6
7       16.0     40.2     60.2     20.0     39.9
8       60.0     20.0     40.0     59.9     20.0
9       59.8     60.0     40.2     20.0     59.8
10      60.0     59.9     60.1     60.0     40.2
11      20.0     40.0     60.0     20.1     40.0
12      60.0     20.4     40.0     59.6     20.0
13      59.9     60.0     40.2     20.0     59.8
14      60.0     59.9     60.1     60.0     40.2
15      20.0     40.0     60.0     20.0     40.0
16      60.0     20.0     39.9     60.0     20.1
17      59.8     60.0     40.2     20.0     59.8
18      60.0     59.9     60.1     60.0     40.2
19      20.0     40.1     60.0     20.0     39.9
20      60.0     20.0     40.0     59.9     20.4
21      59.5     60.0     40.2     20.0     59.8
22      60.0     59.9     60.1     60.0     40.2
23      20.0     40.1     60.0     20.1     40.0
24      60.0     20.0     40.0     60.1     20.0
25      59.8     60.0     40.2     20.0     59.9
26      60.0     59.9     60.1     60.0     40.2
27      20.0     40.0     60.0     20.0     40.0
28      60.0     20.0     39.9     60.0     20.0
```

```
29    59.8   60.0   40.2   20.0   59.8
30    60.3   59.5   60.1   60.0   40.2
31    20.0   40.1   60.0   20.0   39.9
32    60.0   20.0   40.0   59.9   20.0
33    59.8   60.0   40.2   20.0   59.8
34    60.0   59.9   60.1   60.0   40.2
35    20.3   39.7   60.0   20.1   39.9
36    60.0   20.0   40.0   60.0   20.0
37    59.9   60.0   40.2   20.0   59.9
38    60.0   59.9   60.1   60.0   40.2
39    20.0   40.0   60.0   20.0   40.0
40    60.0   20.0   40.0   60.0   20.0
41    59.8   60.0   40.2   20.0   59.8
42    60.0   59.9   60.1   60.0   40.2
43    20.0   40.1   60.0   20.1   39.9
44    60.0   20.3   39.6   59.9   20.0
45    59.9   60.0   40.2   20.0   59.8
46    60.0   59.9   60.1   60.0
```

F.2.3 Verification results for random10

```
1     Range: 15, Track: 18, Standby: 18, Reliable: 24, Unreliable: 24
2     Events processed: 99, Operations checked: 1562
3     Assertions tripped: 0, Verification failures: 0
4
5     Intervals between events (ms):
6
7        37.1   15.5   40.1   20.3   39.9
8        20.0   39.9   20.1   39.9   20.0
9        40.0   20.1   39.9   20.2   39.8
10       20.0   39.9   20.1   39.9   60.0
11       20.2   39.8   19.9   40.0   20.1
12       39.9   20.1   39.9   20.1   40.0
13       20.0   39.9   20.0   40.0   20.1
14       39.9   20.1   40.0   20.1   39.8
15       60.0   60.0   20.1   39.9   59.9
16       20.2   39.9   59.9   20.1   59.9
17       59.9   40.1   20.1   59.8   60.0
18       59.9   60.0   60.0   60.0   40.1
19       20.2   39.9   20.0   40.3   19.8
20       40.3   19.8   39.9   20.0   40.0
21       20.1   39.9   20.1   39.9   20.0
22       39.9   20.1   39.9   60.0   20.2
23       39.9   20.0   40.0   20.2   39.9
24       20.1   39.9   20.1   40.0   20.0
25       39.9   20.0   40.0   20.1   39.9
```

```
26      20.1    40.0    20.1
```

F.2.4 Verification results for random20

```
1     Range: 30, Track: 36, Standby: 36, Reliable: 48, Unreliable: 48
2     Events processed: 198, Operations checked: 3201
3     Assertions tripped: 0, Verification failures: 0
4
5     Intervals between events (ms):
6
7       24.3    37.0    19.2    40.0    20.1
8       39.7    20.3    39.6    20.3    39.8
9       20.3    39.8    20.3    39.7    20.3
10      39.6    20.3    39.7    20.3    59.9
11      39.9    20.3    39.6    20.3    39.7
12      20.3    39.7    20.3    39.8    20.3
13      39.6    20.3    39.7    20.3    39.7
14      20.2    39.8    20.3    39.7    20.4
15      59.8    60.0    39.8    20.4    59.9
16      39.9    20.4    59.8    39.9    60.0
17      60.3    19.9    39.8    59.9    60.0
18      60.0    60.0    60.0    60.0    20.4
19      39.7    20.3    39.6    20.3    39.7
20      20.4    39.6    20.4    39.7    20.3
21      39.7    20.8    39.3    20.4    39.6
22      20.3    39.7    20.4    60.1    39.5
23      20.4    39.6    20.3    39.8    20.4
24      39.6    20.3    39.7    20.4    39.5
25      20.3    39.8    20.2    39.8    20.3
26      39.7    20.4    39.5    60.0    20.4
27      39.7    20.4    39.6    20.4    39.7
28      20.3    39.5    20.3    39.8    20.3
29      39.7    20.4    39.6    20.4    39.6
30      20.4    39.6    20.4    59.8    40.1
31      20.2    39.6    20.3    39.7    20.4
32      39.6    20.3    39.8    20.4    39.5
33      20.4    39.7    20.3    39.7    20.3
34      39.7    20.4    39.6    20.4    59.7
35      60.0    39.8    20.4    59.7    39.9
36      20.3    59.8    39.9    60.0    59.9
37      20.3    39.8    59.9    60.0    60.0
38      60.1    60.0    60.0    20.4    39.7
39      20.4    39.6    20.3    39.8    20.2
40      39.6    20.3    39.7    20.2    40.1
41      20.0    39.6    20.3    39.7    20.3
42      39.7    20.4    59.7    39.9    20.3
```

43	39.6	20.3	39.8	20.4	39.6
44	20.3	39.8	20.4	39.5	20.3
45	39.8	20.3	39.7	20.3	39.7
46	20.4	39.6			

VITA

Surname: O'Connell Given Names: Gordon Wayne

Place of Birth: New Westminster, British Columbia, Canada
Date of Birth: October 6, 1953

Educational Institutions Attended:

University of Victoria	1991 to 1997
University of Victoria	1988 to 1991
University of the Pacific	1977 to 1979
University of Victoria	1973 to 1976

Degrees Awarded:

B. Sc.	University of Victoria	1991
B. Sc (Honours)	University of Victoria	1976

Publications:

C. D. Levings, E. P. Anderson and G. W. O'Connell. "Biological Effects of Dredged-Material Disposal in Alberni Inlet." In B. Ketchum, J. Capuzzo, W. Burt, I. Duedall, P. Park and D. Kester, editors, "Wastes in the Ocean, Volume 6: Near-Shore Waste Disposal", pages 131–155. John Wiley and Sons, New York, 1985.

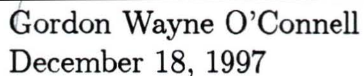
PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

An Implementation of the State Transition Event Model

Author:



Gordon Wayne O'Connell
December 18, 1997