NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

USING DISCRETE-EVENT SIMULATION TO ADDRESS THE PROBE EFFECT IN SOFTWARE TESTING OF REAL-TIME DISTRIBUTED SYSTEMS

by

Robert Milton Ollerton

September 1998

Thesis Advisor: Timothy Shimeall

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson			
1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE September 1998	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE USING DISCRETE-EVENT SIMULATION TO ADDRESS THE PROBE EFFECT IN SOFTWARE TESTING OF REAL-TIME DISTRIBUTED SYSTEMS			5. FUNDING NUMBERS
6. AUTHOR(S) Ollerton, Robert Milton			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES) SPAWAR Systems Center 53560 Hull Street San Diego, CA 92152-5001			10. SPONSORING/ MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The term *probe effect* denotes behavioral changes caused by introducing delays into a concurrent program with synchronization errors. This thesis investigates the feasibility of developing discrete-event simulation (DES) models of software architectures to perform software testing free of the probe effect.

A message-passing subsystem (MPS) and simulated MPS (SMPS) were developed in Java that runs with the same application code. An MPS platform-performance model (MPPM) was developed using dual-loop benchmarking and was integrated into the SMPS. Two demonstration programs were developed to study SMPS timing and its model of a preemptive multi-threaded run-time system. The SMPS-based program behavior was compared to hypothetical execution on a platform with a perfect system clock and no execution overhead.

The differences between hypothetical and observed SMPS-based execution were found to correctly reflect the MPPM. The results indicated that it is feasible to develop DES implementations of some software architectures to perform software testing.

14. SUBJECT TERMS Software testing, object-oriented analysis, Java, discrete-event simulation, dual-loop benchmark, software architecture, real-time, distributed systems			15. NUMBER OF PAGES 112
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT Unlimited

USING DISCRETE-EVENT SIMULATION TO ADDRESS THE PROBE EFFECT IN SOFTWARE TESTING OF REAL-TIME DISTRIBUTED SYSTEMS

Robert Milton Ollerton
A.B., San Diego State University, 1982

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 1998

Author: Robert Milton Ollerton
Robert Milton Ollerton

Approved By: Timothy Shimeall
Timothy Shimeall, Thesis Advisor

Larry J Peterson
Larry Petersen, Second Reader

Dan C Boger
Dan Boger, Chairman,
Department of Computer Science

ABSTRACT

The term *probe effect* denotes behavioral changes caused by introducing delays into a concurrent program with synchronization errors. This thesis investigates the feasibility of developing discrete-event simulation (DES) models of software architectures to perform software testing free of the probe effect.

A message-passing subsystem (MPS) and simulated MPS (SMPS) were developed in Java that runs with the same application code. An MPS platform-performance model (MPPM) was developed using dual-loop benchmarking and was integrated into the SMPS. Two demonstration programs were developed to study SMPS timing and its model of a preemptive multi-threaded run-time system. The SMPS-based program behavior was compared to hypothetical execution on a platform with a perfect system clock and no execution overhead.

The differences between hypothetical and observed SMPS-based execution were found to correctly reflect the MPPM. The results indicated that it is feasible to develop DES implementations of some software architectures to perform software testing.

TABLE OF CONTENTS

I. INTRODUCTION 1
A. RESEARCH QUESTIONS 1
B. METHODOLOGY 2
C. OVERVIEW OF THE THESIS 3
II. BACKGROUND 5
A. THE PROBE EFFECT 5
B. DISCRETE-EVENT SIMULATION 9
C. OBJECT-ORIENTED ANALYSIS 11
D. DEFINITION-USE PATHS WITH RESPECT TO THE CLOCK 15
E. APPROACH 17
III. THE MESSAGE-PASSING SUBSYSTEMS 19
A. TERMINOLOGY AND GRAPHICAL NOTATION 19
B. APPLICATION-LEVEL VIEW OF THE MPS AND THE SMPS 21
C. THE SMPS MODEL OF THE MPS 27
D. SUMMARY 38
IV. MPS PLATFORM PERFORMANCE MODELLING 41
A. MPPM REQUIREMENTS 41
B. DUAL-LOOP (DL) TESTING OVERVIEW 42
C. MPPM DEVELOPMENT 46
D. GENERAL DL TEST DESIGN AND IMPLEMENTATION 48
E. INCORPORATING THE MPPM INTO THE SMPS 49
F. CONCLUSIONS 51
V. DEMONSTRATION PROGRAMS 53
A. DINING PHILOSOPHERS (DP) 53
B. PERIODIC HARMONIC TASK SET WITH RMS PRIORITY ASSIGNMENT 58
C. SUMMARY 60
VI. CONCLUSIONS AND RECOMMENDATIONS 63
A. SUMMARY AND CONCLUSIONS 63
B. RECOMMENDATIONS 66
LIST OF REFERENCES 69
APPENDIX A. MPS AND SMPS OOA MODEL DIAGRAMS 73
APPENDIX B. MPPM SOURCE CODE EXAMPLES 83
APPENDIX C. DINING PHILOSOPHERS OOA MODEL DIAGRAMS 89
APPENDIX D. DINING PHILOSOPHER PROGRAM EXECUTION TRACES 95
INITIAL DISTRIBUTION LIST 101

ACKNOWLEDGEMENT

It has been a privilege to work with my advisor, Dr. Timothy Shimeall. He allowed considerable independence and gave strong guidance when each was needed the most. I am fortunate to have had the opportunity to work with him. I would also like to express my appreciation to my second reader, Dr. Larry Peterson. Dr. Peterson devoted time and attention to detail under tight time constraints that was above and beyond the call of duty.

I would also like to thank Dan Roy, Neil Altman, Patrick Donohoe, Gary Chastek, and Gene Riddel of the Software Engineering Institute for exposing me to software benchmarking technology. I am also particularly grateful to my wife, Anna Noah, for her patience and encouragement throughout this experience.

I. INTRODUCTION

Concurrent programs are used increasingly in modern software systems. Modern high-level programming languages such as Java and Ada have facilities for concurrency control built into the language. This provides the opportunity to develop transportable concurrent programs. Such programs must often meet timing requirements and are typically more complex than their sequential counterparts. More frequent design and implementation errors usually accompany additional complexity. More and better software testing can reduce this problem. However, invasive testing techniques that save or display program state can change timing in concurrent programs by consuming additional CPU time and by altering synchronization relationships with operations such as IO blocking, preemption, or breakpointing. The term *probe effect* denotes behavioral changes caused by introducing delays into a concurrent program [Gait 1986; McDowell and Helmbold 1989; Fidge 1993; Shutz 1993]. The probe effect presents a serious challenge to software testers since it can cause a program to generate different results depending on whether it is being tested.

A. RESEARCH QUESTIONS

The thesis is concerned with two questions related to avoiding the probe effect in testing real-time software implemented in high-level languages (HLL) developed using Object-Oriented Analysis (OOA). The uniprocessor question is whether it is feasible to develop a simulation model of an application-independent software-architecture and environment in which to execute the same (HLL) application-level binary code as would execute in the target environment. The distributed-systems question is whether it is feasible to develop such an approach to support execution of distributed real-time applications.

The capability to execute the same HLL application code in both a simulation model and in its target environment would enable application-level software testing to be conducted under controllable conditions free of the probe effect. Implementations of the same software architecture (SA) that generate the same program results with the same application-level binary-code have application-binary-code consistency (ABCC). The requirement that the same application code execute in both environments implies that both implementations have ABCC and that they have the same interface. If the software-architecture were free of application-content, it would yield a reusable software-architecture for testing and executing real-time applications. Having a capability that relied only on a single HLL for timing and synchronization would produce highly portable and testable software. Both the simulation model and the application software for the target system would be independent of the operating system and hardware platform.

The capability to execute distributed real-time applications in both environments supports the testing of multiple communicating application instances under conditions that are free of the probe effect. This would enable software testers to cover previously unattainable test cases by increasing the amount of control they have over the test environment for distributed systems.

B. METHODOLOGY

As part of the investigation presented in this thesis, two implementations of the same software-architecture were developed. The first is an object-oriented, multi-threaded, asynchronous, message-passing subsystem (MPS). The second is a sequential, discrete-event simulation (DES) model of the MPS (SMPS).

Two design goals drove the development of MPS and SMPS. First, they must have the same interface so that applications, if properly implemented, can execute unmodified on both. The SMPS must also have the same message transmission and reception order as the MPS to the extent that the MPS run-time environment is deterministic.

Since timing characteristics are largely a function of the run-time environment, the SMPS was parameterized to accept a platform-performance model. A platform performance-model acquisition method was developed using a dual-loop measurement technique [Clapp et al. 1986; Altman and Weiderman, 1987; Vestal, 1990] to acquire language feature execution time estimates.

Two demonstration programs were also developed. An implementation of the "Dining Philosophers" program [Dijkstra 1971] was developed to test the timing and synchronization features of the simulation model. The second is an MPS-based program with Rate Monotonic Scheduling (RMS) priority assignments and artificial workloads [Sha and Goodenough, 1990]. This program was developed to study SMPS timing and the model of a preemptive multi-threaded run-time system.

C. OVERVIEW OF THE THESIS

Chapter II of the thesis presents background material consisting of brief overviews of related approaches to addressing the probe effect, relevant aspects of DES modelling, and OOA. Chapter III describes the operational models and design of the MPS and SMPS and addresses issues related to SMPS fidelity. Chapter IV describes platform performance-model development. Chapter V presents descriptions of the demonstration programs and discusses their results and implications for distributed systems. Finally, Chapter VI presents the conclusions, discusses relevance to other application areas, and presents recommendations for further research.

Appendix A contains the OOA model diagrams for the MPS and the SMPS and sample OOA model diagrams for a simulated distributed MPS (SDMPS). Appendix B contains MPS and SMPS source code examples. Appendix C contains the "Dining Philosopher" OOA model diagrams. Appendix D contains selected sections of the execution traces for the Dining Philosophers and RMS programs.

II. BACKGROUND

This chapter provides background on approaches to addressing the probe effect and explains its significance for software testing. It also provides background on simulation modelling and defines the approach used in the current investigation. Finally, it describes the derivative of OOA used in the development approach.

A. THE PROBE EFFECT

Probe effect delays are delays caused by intrusive techniques such as running the program in a debugger and by source code instrumentation in testing. Delays may also be introduced by operating system effects [Gait 1986] and by other sources of non-determinism such as communication protocols [Tannenbaum 1996]. Gait states that the probe effect often manifests itself in two ways [Gait 1986]. Either a non-functioning program begins to operate as desired when delays are inserted or an apparently working program yields incorrect results with inserted delays. There are also two other ways the probe effect can manifest itself. One is when a program that previously gave incorrect results yields different incorrect results with inserted delays. The other is when a functioning program gives different correct results with inserted delays. Although this thesis is concerned with the probe effect in software testing, most of the other work to eliminate or mitigate the probe effect addresses its role in debugging.

Netzer and Miller state that two messages in a message-passing parallel program can race if they are simultaneously in transit and either could arrive first at some receive operation [Netzer and Miller 1994]. Lamport defines the *happened before* relation and *concurrent* as follows [Lamport 1978]:

> *Definition.* The relation "$\rightarrow$" on the set of events of a system is the smallest relation satisfying the following three conditions: (1) If a and b are events in the same process, and a comes before b, then $a \rightarrow b$. (2) If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$. (3) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. Two distinct events a and b are said to be concurrent if $\neg(a \rightarrow b)$ and $\neg(b \rightarrow a)$.

Netzer and Miller's *racing messages* are examples of Lamport's *concurrent events*. A synchronization error exists if the arrival order of two racing messages affects program correctness [Netzer and Miller 1994]. The probe effect adds to the risk that actual test cases may not implement test case design by affecting the arrival order of racing messages.

When a race is detected, the logical time of the message receipt is traced, or *recorded*. Traced times accompany *racing* messages and during re-execution, message receipt for racing messages is restricted to the traced time from a previous execution. This approach is more efficient than other *trace and replay* schemes that constrain all message receptions to traced times. However, it maintains the same synchronization properties as the original execution by preserving the *happened before* relation [Netzer and Miller 1994].

Jain et al. used intrusive techniques to monitor timing behavior for real-time systems [Jain et al., 1996]. The approach maintains *local time* as the value pair (CT, Δ), where CT is the *clock time* obtained from the internal clock, and Δ is the current *intrusion time*. In this case intrusion time is the CPU time consumed by executing the monitoring instructions on the same CPU. The estimate of clock time, given as CT-Δ, replaces clock time in timing computations. This prevents the probe effect from affecting program results. This approach uses a model of actual clock time and requires accurate knowledge of the timing properties of monitoring operations.

Reproducibility, like the term *test repeatability*, describes whether a program computes the same results when repeatedly executed with the same inputs [Schutz 1993]. One way to avoid the probe effect in debugging is to permanently install debugging probes in the program so the program undergoing debugging is the same as the final version [McDowell and Helmbold 1989; Fidge 1993]. Practical debugging problems attributed to the probe effect are often manifestations of the difficulty of achieving reproducibility [Fidge 1993].

In Timewarp, there is one local virtual clock per process and messages to be sent are timestamped with the virtual time when they must be received. Local virtual time is updated to the timestamp of the next event in the process input queue. When a process receives a message, *m*, with a receive-time that is earlier than its local virtual time, its state and local virtual time are restored to *m*'s receive-time and any synchronizing actions it performed since that time are undone [Jefferson 1985].

Tolmach and Appel developed an interactive debugger with reverse execution for the language Standard ML [Tolmach and Appel 1991]. This debugger associates the value of a software instruction counter, a so-called *s-time*, that is incremented each time a debugger breakpoint or periodic checkpoint is encountered. Since this approach was highly vulnerable to the probe effect, they proposed a model of concurrency debugging based on the Timewarp system [Jefferson 1985]. This proposal includes the concept of a virtual multiprocessor consisting of a number of virtual processors. Each virtual processor would be characterized by an execution rate and its current s-time would be substituted for Timewarp-style local virtual time. The system would compensate for probe effects by rolling back operations that violate the total ordering of synchronizing operations as defined by s-times. By varying the specified number and execution rates of virtual processors one could obtain data from different program runs that would help expose time-dependent behavior

[Tolmach and Appel 1991].

Huang developed a software simulator of a 4MHz Z80 processor designed to execute programs in machine code to facilitate testing and debugging of real-time programs [Huang et al. 1983]. This avoided the problem that traditional debuggers have of handling interrupts. It also avoids the problem that real-time simulators have of being required to process large quantities of mostly irrelevant data. This is another example of an approach that uses a model of time to avoid the probe effect. Similar approaches are often used for testing real-time control systems.

The significance of the probe effect and unpredictable delay insertion for software testing is the adverse effect on observability and controllability [Shutz 1993]. The probe effect adds to the risk that actual test cases may not implement test case design. The goal of software testing is to find errors and a successful software test is one that reveals an error [Myers 1979]. Debugging has been described as diagnosis and correction performed after executing a successful test case [Myers 1979; Shutz 1993]. The impact of the probe effect on software testing is more closely related to controllability than reproducibility, but the reverse is true for debugging.

Gupta's approach accurately reports timing information by compensating for probe effect intrusion time [Gupta et al. 1996]. However, the approach does not address reproducibility or controllability needed for testing. Netzer and Miller's approach requires the development of a debugger. This makes portability costly [Netzer and Miller 1994]. Tolmach and Appel's approach also has limited portability because it is restricted to the Standard ML language. Huang's approach requires development of a virtual processor for each required processor type [Huang et al. 1983], also making portability a difficult problem. It require access to the operating system machine-code for each target.

The approach presented in this work assumes the existence of two versions of a tested and debugged application-independent software-architecture and a performance model generation program. Testing the software on another platform should only require generation of a new performance model. Actual testing can be performed on any platform that can support the simulation model using the newly acquired performance model as input.

B. DISCRETE-EVENT SIMULATION

Russell defines simulation as follows: "A simulation of a system is the operation of a model that is a representation of the system" [Russell 1983]. Models of systems may be continuous or discrete depending on the model of changes in system state [Graybeal 1980]. Models of systems may also be stochastic or deterministic. Stochastic models contain a certain amount of randomness. In deterministic models the system evolves completely deterministically from one state to the next [Graybeal 1980].

Most discrete simulation tools have either a time-driven or an event-driven clock (EDC) policy. A time-driven policy usually consists of repeatedly incrementing the clock with a fixed time quantum and then servicing all events scheduled to occur at that simulation time. In such cases, the time quantum is the clock granularity. An EDC policy usually can be implemented by repeatedly removing the event from the event set with the earliest activation-time, updating the clock to the time of that event, and then servicing that event. Events are often serviced by executing an *event-subprogram*. An event-subprogram is a subprogram that models the behavior of one or more events occurring at an instant in time. The Java method named *current.activate* shown in Fig. 2.1 is an example of an event-subprogram.

DES models may also be process-oriented or event-activation-oriented.[1] SIMSCRIPT II.5 [Russell 1983], Simula 67 [Birstwistle et al. 1973], and the Ada Process-Oriented Simulation Library (APOSL) [Ollerton 1992] are examples of process-oriented DES systems. Process-oriented systems have a concept of a *process* and may have state and a lifetime. The process reacts to events that may cause it to perform an action, simulate the passage of time, become suspended or interrupted, and so on. In an event-activation-oriented system *instantaneous* events occur at scheduled times in a model. Occurrences of such events and the separations in time between them can be functionally equivalent to the life-cycle of a process in a process-oriented system.

In general, process-oriented approaches require more resources than event-activation-oriented approaches, but processes are very convenient modelling abstractions for entities with life-cycles. However, they generally consume more memory and CPU time than event-activation-oriented approaches due to the necessity to save and switch execution context during process state transitions.

In a DES model, a positive difference in time between the occurrence of two successive events represents the passage of time. It is commonly used to represent the duration of an action. A *DES-action* represents activity occurring over a finite positive amount of

```
  public static void EDC_Loop(){
    current = nextEvent();
    while (current != null){
       clock = clock + current.activation_time;
       current.activate();
       current = nextEvent();
    }
}
```

Figure 2.1. Example of a Java Implementation of an EDC Policy

1. The term event-activation-oriented is used since there does not appear to be a single commonly used term to describe this approach.

time modelled by the occurrence of two events denoting the beginning and end of the action. *DES-action-time* is the difference between the activation-times of these events. The granularity of DES-action-time impacts model fidelity, performance, and development effort of DES simulation models.

During this work a minimal event-activation-oriented DES library with an EDC policy was developed in Java to support a deterministic model of the MPS running in a prioritized, preemptive, multi-threaded environment.

C. OBJECT-ORIENTED ANALYSIS

Rumbaugh states that the term *object-oriented* means organizing software as a collection of discrete objects that incorporate both data structure and behavior [Rumbaugh, et al., 1991]. *Inheritance* and *polymorphism* are two properties that are also associated with object-orientation. *Inheritance* refers to the object-oriented property that when descendents of types are defined through type extension or specialization [Rumbaugh et al. 1991] they inherit the characteristics of the ancestor type. *Polymorphism* describes the property of descendents having different implementations of an operation that is defined in the ancestor type. Different implementations of a polymorphic operation must have the same signature, that is, the same number and types of arguments and the same result type [Rumbaugh et al. 1991].

The term *Object-Oriented Analysis* was first used to denote the Shlaer and Mellor method (SMM) "for identifying the significant entities in a real-world problem and for understanding and explaining how they interact with one another" [Shlaer and Mellor 1988]. This consists mainly of producing a set of information, state, and process models by applying the rules of the method [Shlaer and Mellor 1988; Lang 1993; Shlaer and Lang1996].

The term *OOA* is now commonly used to denote object-oriented high-level design and analysis methods for software engineering.

The SMM distinguishes itself from other OOA and object-oriented design approaches in three ways. First, objects are abstractions of real-world things in the application domain as opposed to representing conceptual entities of the design [Shlaer and Mellor 1997]. Second, analysis proceeds by separating systems into various domains, or subject areas, that are analyzed separately. These are the application domain, various service domains, the software-architecture-domain, and other architecture-domains. Finally, SMM is a translation-based method, as opposed to being elaborative [Bell 1998]. In an elaborative method such as *Object Modelling Technique* (OMT), object design is a process of adding detail to existing models and making implementation decisions [Rumbaugh et al. 1991]. SMM translation is a more elaborate process.

SMM translation requires application analysis, architecture, an application-instance-to-architecture-instance mapping, and a system-construction engine. SMM analysis "consists of work products that identify the conceptual entities of a single domain and explain, in detail, the relationships and interactions between these entities" [Shlaer and Mellor 1997]. These consist primarily of a domain model, an object information model (OIM) for each domain, a state-model for each *active* object in each OIM, and an action data-flow diagram (ADFD) for each action of each state-model. An *active* object is an object with interesting behavior. Architecture deals completely with the following concerns:

- policies and mechanisms for organizing and accessing data;
- control strategies, including synchronization, concurrency, interactions in a distributed system, and the like;
- structural units--the tasking strategy for the system, strategies for allocating

tasks to processors and processing units to tasks, standard structures used with a task; and

- time--mechanisms for keeping time and for creating delayed transfers of control. [Shlaer and Mellor 1997].

Archtypes are much like macros and provide the application-instance-to-architecture-instance mapping. "An archtype combines text, written in the target programming language, with placeholders used to represent information from the architecture's instance database" [Shlaer and Mellor 1997]. An instance database is a database containing information about preexisting instances from the associated domain. A system-construction engine is "a script... that will generate the code for the system from the analysis models, the archtypes, and the instance databases of all domains" [Shlaer and Mellor 1997].

SMM translation requires an architecture-independent analysis expressed in complete detail and compliant with the rules [Shlaer and Mellor 1988; Lang 1993; Shlaer and Lang 1996] of the method [Shlaer and Mellor 1997]. It also requires an application-independent architecture also expressed in complete detail in both conceptual and archtype terms" [Shlaer and Mellor 1997]. The system-construction engine takes the instance databases and analysis models as input and expands the archtypes to produce the code.

The SMM method is relatively formal [Shlaer and Mellor 1988; Lang 1993; Shlaer and Lang1996] and requires a considerable investment of effort to produce a complete set of archtypes for the architecture-domain and a complete set of models for multiple domains. This is likely to be more expensive than elaborative approaches for one-time development efforts. It is also likely to be relatively expensive when applied to problems with volatile requirements. In his summary on translative approaches, Bell commented that "The key determinant is the potential return from reusing the architectures" [Bell 1998].

A complete application of the SMM is impractical for this effort. The models presented in this work are primarily for descriptive purposes. However, the development approach used in this thesis is similar to the SMM in the following ways. First, it views application and architecture as separate analysis problems. Second, objects represent conceptual entities in the analysis domain, as opposed to design entities. Finally, it uses state machines, as opposed to OMT state-charts, to model system dynamics.

The SMM defines *action time* as the time required to execute a state-model action [Shlaer and Mellor 1992]. In order to avoid confusion with DES-actions and DES-action-time, state-model action and state-model action time will be referred to from here on as *SM-action* and *SM-action-time*.

Huang's Z80 simulator referred to in section A above [Huang et al. 1983] could have been implemented as a DES model with an EDC policy and default granularity of one CPU cycle. The three main capabilities required to achieve this are the capabilities to fetch the next instruction, to execute the instruction, and to update the clock. Figure 2.2 shows how a hypothetical EDC-loop could be implemented in Java to achieve this.

This approach could also be viewed as state machine execution. Each SM-action corresponds to a DES-action and SM-action-time corresponds to DES-action-time from this viewpoint. This correspondence is highly desirable because it allows the application

```
 public static void Z80_Loop(){
   current = fetchInstruction();
   while (current != null){
      current.execute();
      clock = clock + current.clock_cycles;
      current = fetchInstruction();
   }
}
```

Figure 2.2. Java Code Fragment of a Hypothetical Z80 Simulator

source code to execute in the simulator unchanged. However, it also requires more EDC-loop overhead than the latter approach.

The portability problems mentioned in section A limit the usefulness of machine-code-level modelling of programs written in HLLs with built-in concurrency such as Java or Ada. Unfortunately, HLL statement-level modelling for such languages is likely to be unsatisfactory for concurrent programs. It would raise the level of abstraction and achieve greater portability at the expense of model fidelity because single HLL statements can mask complicated interactions.

D. DEFINITION-USE PATHS WITH RESPECT TO THE CLOCK

Figure 2.3 graphically depicts the views of time in SM-action and DES-action execution of the same *n* statements. In each case, the value of the internal clock is t_n after *n* statements have completed execution. However, after statement s_0 has executed, the value of the internal clock is t_1 in the SM-action and is still t_0 in the DES-Action. In the SM-action, each statement is viewed as consuming a quantity of time as indicated in the *time* col-

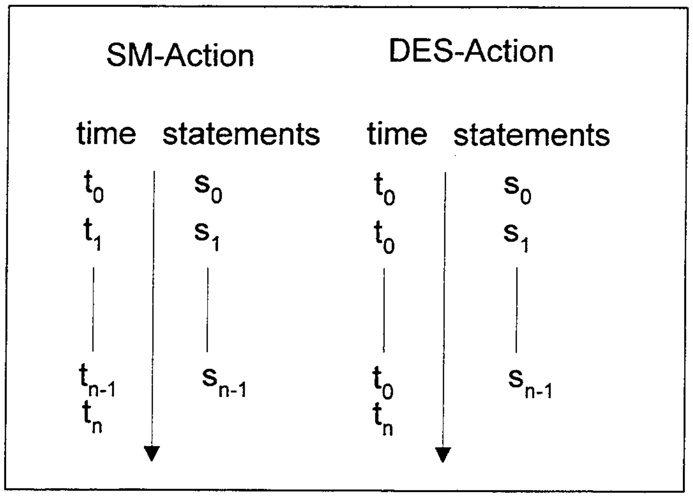

Figure 2.3. Correspondance of Real Time and Simulation Time in SM-action and DES-Action Execution

umn for the SM-action. However, the DES-action time column shows that no simulation time elapses between statement executions in the DES-action.

Clarke defines a *module* to be either a main program or a single subprogram that has only one entry and one exit point and defines *control flow graph* as follows.

> A *control flow graph* of a module M is a (not necessarily unique) directed graph $G(M) = (N, E, n_s, n_f)$, where N is the (finite) set of nodes, $E \subseteq N X N$ is the set of edges, $n_s \in N$ is called the *start node*, and $n_f \in N$ is called the *final node*. Each node in N, except the start node and the final node, represents a statement fragment in M, where a statement fragment can be a part of a statement or a whole statement. [Clarke et al. 1989]

Clarke defines *PATH(M)* as the set of all paths in *G(M)* [Clarke et al. 1989]. Jorgenson refers to "a program P that has a program graph G(P), and a set of program variables V" in the following four definitions [Jorgenson 1995].

> Node n $\in$ G(P) is a *defining node* of the variable v $\in$ V, written as DEF(v,n), iff the value of the variable v is defined at the statement fragment corresponding to node n.
>
> Node n $\in$ G(P) is a *usage node* of the variable v $\in$ V, written as USE(v,n), iff the value of the variable v is used at the statement fragment corresponding to node n.
>
> A *definition-use (sub)path* with respect to a variable v (denoted du-path) is a (sub)path in PATHS(P) such that, for some v $\in$ V, there are define and usage nodes DEF(v,m) and USE(v,n) such m and n are the initial and final nodes of the (sub)path.
>
> A *definition-clear (sub)path* with respect to a variable v (denoted dc-path) is a definition-use (sub)path in PATHS(P) with initial and final nodes DEF(v,m) and USE(v,n) such that no other node in the (sub)path is a defining node of v [Jorgenson 1995].

The discrepancy between the views of time in SM-actions and DES-actions reflects the fact that the internal clock is continually updated by the hardware in the target system and is managed by the EDC policy implementation in the DES system. Every du-path with respect to the simulation-clock in an EDC DES-action is a dc-path whose defining node is the clock-update-operation that preceded the action. Every usage node with respect to the internal clock in a SM-action on the target platform must be viewed as though there were an immediately preceding defining node for the internal clock in the program graph.

This is not a problem for the DES-actions in the hypothetical model depicted in Fig. 2.2. In that model, the clock update operation immediately precedes the single statement in the DES-action. However, it is a problem for HLL statement-level modelling because a single HLL statement can mask machine-code-level branching and preemption by higher priority tasks.

E. APPROACH

This thesis avoids the problem caused by HLL statement-level modelling by employing two related strategies. First, it delegates synchronization and timing operations to the software-architecture-domain and imposes policies that prohibit application-domain access to HLL statements for timing and synchronization. This allows the two implementations of the software-architecture, the MPS and the SMPS, to have the same interface. Second, it prohibits application-domain access to the internal clock and simulation clock, and provides access to logical clocks instead. The logical clock is updated to the value of the internal clock just prior to SM-action invocation, so every du-path with respect to the internal clock is a dc-path whose defining node is the clock-update-operation that preceded the action. This assures that application-domain du-paths with respect to logical clocks are the same in the SM-actions as in the DES-actions.

III. THE MESSAGE-PASSING SUBSYSTEMS

The first section of this chapter defines the terminology and graphical notation used to describe the MPS and SMPS. The second section describes the MPS and SMPS from a single application-level viewpoint. It also describes the sufficient conditions to achieve the goal of assuring that the application code running with the MPS exhibits the same behavior when running with the SMPS. The third section describes the SMPS model of MPS operation in detail.

A. TERMINOLOGY AND GRAPHICAL NOTATION

There is no standard terminology to distinguish between conceptual entities of object-oriented analysis (OOA), object-oriented design (OOD), and object-oriented programming (OOP). This thesis follows SMM convention by using the term *object* to refer to a set of real-world things at the analysis level [Shlaer and Mellor 1992]. Individuals in the set are object-*instances*. *Design* describes the approach to implementing the application specified in the analysis. This thesis also employs the SMM convention of using the term *class* to refer to a set of entities at the design level [Shlaer and Mellor 1992]. Individuals in the set are referred to as class instance*s* or class *members*. Every design class is implemented in a corresponding Java *class.* The term *member-data* is used to refer to the data associated with a class member. The term *class-data* is used to refer to data that is global to all instances of a class.

A number of OOA diagrams are presented in the appendices to facilitate the descriptions of system structure and function. These diagrams use OMT graphic notation [Rumbaugh 1991] but use SMM semantics [Shlaer and Mellor 1992].

Objects in the object-information model (OIM) diagrams are depicted as boxes with two compartments. The top compartment contains the object name, and the bottom compartment contains object attributes. Attributes with an asterisk are object identifiers. A single asterisk denotes a primary identifier and more than one asterisk denotes an auxiliary identifier. An attribute followed by "(R*n*)," where *n* is some number, is a relational attribute that formalizes a relationship.

Object relationships are bidirectional and with the exception of subclass-superclass relationships, are depicted as connectors with arrowheads on each end. Clear and black arrowheads denote conditional and unconditional participants in relationships. Multiplicity is indicated by one or two arrowheads on an end of a relationship connector. One arrowhead indicates a single participant and two arrowheads indicates many participants. Relationship R7 in Fig. A-1 depicts a one-to-many conditional relationship that shows that each instance of Activator may be associated with zero or more instances of UnscheduledMessage. It also specifies that each instances of UncheduledMessage is always associated with an instance of Activator.

The subclass-superclass relationship is depicted by a triangle and a set of connectors. The connectors emanating from the base of the triangle are attached to the subclasses. The connector emanating from the apex of the triangle is attached to the superclass. Relationship R5 in Fig. A-1 depicts a subclass-superclass relationship. The UnscheduledMessage and ScheduledMessage objects are subclasses of MpsMessage. Each of these inherits the attributes of the ScheduledMessage superclass object.

In state-model diagrams, boxes with rounded corners denote states and arrows denote transitions. Transitions are labelled with the event label and event data may be listed between parentheses next to the label. The state action is textually described in pseudocode contained in the state box. An event designation preceded by "%g" denotes *event generation.* Event generation is equivalent to *sending* an event to the state-model of the destination object. The event destination can be inferred from the event name and the pseudocode. For example, the "TH" in the TH4 event in the "Getting Message" state shown in Fig. A-3 indicates that its destination is the thread state model.

In OOA, *delayed-event-delivery* is the delivery of an event to an object's state-model at a specified time. This capability is represented in the SMM formalism by the predefined *timer* object. The *TIM1:set timer* event has an event label, a delivery time, a destination object-instance, and a destination timer-instance. This event causes the timer-instance to send the named event at the specified time to the object-instance. The non-SMM event, *TIM2:cancel,* denotes delayed event-delivery cancellation. *TIM7:fire* denotes timer expiry for a SMM timer.

B. APPLICATION-LEVEL VIEW OF THE MPS AND THE SMPS

1. Overview

The application-binary-code consistency (ABCC) requirement is that the same application binary-code execute in both the target execution-environment and in the simulation model. This investigation used an MPS-based software architecture that mapped one SM-action execution to one MPS message-activation. In the SMPS, each SM-action execution must be implemented with one DES-action for the SMPS to be ABCC with the MPS. This section describes the MPS architecture design-properties that contribute to ABCC.

The MPS and SMPS each comprise a framework to implement OOA models. The frameworks support applications that depend on event-delivery for communication and state-model execution for functionality. Each framework provides capabilities to implement the concepts of active objects, passive objects, synchronous inter-object communication, asynchronous communication, and delayed-event-delivery.

In the MPS operational model, state-machine-execution threads, called activators, generate messages in SM-actions and transfer them to activator message-buffers or to the timer-thread delay-queue. The timer-thread transfers messages in the delay-queue to the appropriate activator message-buffers at the scheduled delivery time for each message. An activator repeatedly acquires the next message from its buffer and executes the appropriate SM-action. However, an activator will wait for a message if none are available.

The MPS and SMPS define two superclasses, *MpsObject* and *MpsMessage*, both shown in Fig. A-1 and Fig. A-2, that provide many of the interfaces needed to implement the information structure and system dynamics specified in an OOA model. Application developers produce specializations of MpsObject and MpsMessage that extend their properties and inherit their interfaces. Specializations of MpsObject contain member-data that correspond to object-attributes defined in the application OIM. Such specializations also contain methods that implement state-model actions for active objects in an application OOA. Direct access to member-data or *get*-methods and *put*-methods implements synchronous inter-object communication specified by accessor-processes in state-model ADFDs.

The activator-objects (AO) in an MPS-based application are the activators, MpsObjects, and MpsMessages shown in Fig. A-1 and Fig. A-2. Execution priority forms a partition on the set of AO in MPS-based applications. The partition is indicated by relationship R1 in Fig. A-1 and Fig. A-2. All MpsObject and MpsMessage methods are restricted to execute in the context of the activator with the priority that defines their equivalence class,

AO_p, with two exceptions. The two exceptions are the methods that implement event-generation and delayed event-delivery. Instances of each MpsMessage subclass may only be sent to the MpsObject subclass, or to instances of the subclass indicated by relationship R4.

Specializations of MpsMessage implement events specified in OOA state-models. The MpsObject superclass defines a *send*-method that takes a parameter of type *UnscheduledMssage* to implement event-generation. UnscheduledMessage-instances are placed in the activator message-buffer depicted in relationship R7. Event data are implemented as MpsMessage member-data.

The *timer*, the *delay-queue*, the MpsObject *schedule*-method, and the *ScheduledMessage* subclass of MpsMessage provide the OOA delayed-event-delivery capability. Relationship R6 in Fig. A-1 and Fig. A-2 depicts the delay-queue and its relationship to the timer. The schedule-method has two arguments, an *activation-time* and an ScheduledMessage-instance. When a schedule-method is invoked, it assigns the activation-time to the ScheduledMessage-instance and inserts the instance into the delay-queue. The timer repeatedly gets the ScheduledMessage-instance with the earliest activation-time, waits until that time is reached, and then transfers the MpsMessage-instance to the appropriate activator message-buffer.

The MpsMessage class has an abstract *activate*-method that must be overridden with a concrete implementation by each concrete MpsMessage subclass. Activate-method execution must determine the next state for the destination MpsObject-instance and execute the SM-action on its behalf. Each activator serially executes SM-actions on behalf of its MpsObject-instances by executing activate-methods.

2. Policies and Properties

Clarke defines a program P with a set of variables V [Clarke et al. 1989]. An activate-method A in program P with control flow graph $G(A) = (N, E, n_s, n_f)$ has initial-state S_i in which $(v \in S_i) \Leftrightarrow (v \in V \wedge n \in N \wedge USE(v,n))$. Activate-method A has a final state, S_f, in which $(v \in S_f) \Leftrightarrow (v \in V \wedge n \in N \wedge DEF(v,n))$. An activate-method that generates the same final state in the MPS and SMPS when given the same initial state has *activate-execution (AE) predictability.*

An SMPS program that executes its activate-methods in the same order in the SMPS as in the MPS has *activate-invocation-order (AIO) consistency.* An MPS-based application will have the same timing and synchronization properties and will produce the same results in the SMPS as in the MPS if the activate-methods are AE-predictable and the SMPS program is AIO-consistent.

Let V be the set of variables v in program P and let "$\rightarrow$" denote the *happens before* relation [Lamport 1978] applied to the execution-order of statements and statement fragments in P. An MPS-based program P will have AE-predictability if for all of P's executions and for any two non-unique activate-methods X and Y in P with $G(X) = (M, G, m_s, m_f)$ and $G(Y) = (L, F, l_s, l_f)$, $DEF(v,l)$ and $USE(v,m) \Rightarrow (m_f \rightarrow l_s \wedge l_f \rightarrow m_s) \vee (l_f \rightarrow m_s \wedge m_f \rightarrow l_s)$.

The MPS and SMPS *data-access policy* is that for any two MpsObject-instances, $O1_p \in AO_p$ and $O2_q \in AO_q$, $p=q \Rightarrow$ synchronous data exchange is allowed, $p \neq q \Rightarrow$ synchronous data exchange is not allowed and must be performed by message passing. The serial nature of activator state-machine execution and the data-access policy assure that each MpsMessage-instance in AO_p will have exclusive access to all of the member-data in AO_p

for the duration of its activate-method. The data-access policy confers AE-predictability on MPS-based applications.

An activate-method that accesses the system clock will violate the data-access policy and may undermine AE-predictability because the value of the system clock will change independently of activate-method execution. This problem is avoided by assigning an *EventClock*-instance to each activator. The *EventClock*-instance is updated to system-time just prior to each activate-method execution and remains fixed for its duration. This is depicted by relationships R9 in Fig. A-1 and Fig. A-2. The MpsObject class defines a *millis*-method that returns the time of the associated EventClock-instance via relationships R2 and R9.

The MPS and SMPS have a *synchronization policy* with two rules. First, the OOA state-models for an application must represent all synchronization requirements. Second, all required synchronization must be implemented by message-passing. This policy prohibits application-level use of HLL statements for timing and synchronization. That would cause variation in execution-time that would not be modelled in the SMPS. An activate-method has *path execution-time (PET) consistency* if the variation in its MPS execution-time only results from preemption by another MPS thread or from executing different sub-paths in G(A). PET-consistent activate-method execution-time depends only on its initial state, the platform performance, and its preemption time.

The SMPS model assumes PET-consistency. The potential AIO-consistency of an SMPS model of an AE-predictable and PET-consistent MPS-application depends on the fidelity of underlying run-time system model, the MPS-overhead model, and the activate-method execution-time model. Model inaccuracy will degrade AIO-consistency.

MpsMessage send-times in the MPS and SMPS will be different if messages are allowed to be sent at any arbitrary time during activate-method execution. The time that an MpsMessage-instance is sent affects the time that its activate-method is invoked. An SMPS application cannot be AIO-consistent if message *send-times* are different in the MPS and SMPS. As can be seen in Fig. 3.1, this results from the different views of time presented by the MPS system clock and the SMPS EDC.

SMPS and MPS message send-times must be constrained to occur at the SM-action end-time. This requirement is very cumbersome for programmers. In order to avoid this, the *send*-method *and schedule*-method only place messages in temporary storage. The activator transfers the messages from temporary storage to the appropriate buffers upon returning from the activate-method. The SMPS will have accounted for activate-method cpu-time consumption and preemption-time by that point. The message transfer takes place in the *Transferring Msgs* action of the activator state-model shown in Fig. A-4.

An SMPS version of an MPS-based application can have the same timing and synchronization properties and can generate the same results if the activate-methods have AE-predictability and the SMPS version has AIO-consistency. An application that complies with the data-access and synchronization policies will have AE-predictable activate-methods. An application must have PET-consistent activate-methods and PET-consistent MPS-infrastructure code to be AIO-consistent. However, an SMPS-application must incorporate an adequate platform-performance model into all activate-methods and into the SMPS infrastructure to be AIO-consistent.

C. THE SMPS MODEL OF THE MPS

In the MPS operational model, MpsMessage-instances are generated in MpsObject SM-actions and are either transferred to activator message-buffers or to the delay-queue. The timer transfers messages in the delay-queue to the appropriate activator message-buffers at the message activation-times. Activators acquire messages from their message-buffers and invoke their activate-methods.

The SMPS must also account for preemption in the MPS to support AIO-consistency. The model of the Java run-time executive (RTE) is depicted in the interactions between the RTE state-model shown in Fig. A-5 and the thread state-model shown in Fig. A-6. The model of the implementation of the activator-thread *run*-method is depicted in the interactions between the activator state-model shown in Fig. A-5 and the thread state-model shown in Fig. A-6. The model of the implementation of the timer-thread run-method is depicted in the interactions between the timer state-model shown in Fig. A-3 and the thread state-model shown in Fig. A-6. The next three subsections describe the SMPS model of the Java implementations of the MPS activator and timer.

1. Thread Context Switching

The MPS activator and the MPS timer behavioral models are split into a generic thread state-model and a role-specific state-model. The thread state-model implements the model of context switching. Dynamic priorities are used to prevent preemption during certain portions of the life-cycles of activators and timers. This reduces the number of states in which context switching can occur.

Context switching is modelled in the thread state-model as a three-state transition from *Ready* to *Switching Context* to *Running* as shown in Fig. A-6. When the RTE sends

TH2 to a thread in the *Ready* state, the thread moves to the *Switching Context* state. When the thread arrives in that state, it schedules another TH2 event to be delivered to itself when the context switching time expires. The TH2 transition from *Switching Context* to *Running* invokes an SM action to send TH2 to its client thread subclass state-model. Context switches in the timer and activator classes are denoted by the TH2 transition in Fig. A-3 and Fig. A-4.

2. The Activator

In order to achieve AIOconsistent, the SMPS model must account for CPU-time consumed by activate-method execution, activator-preemption, MPS overhead, and thread context-switching.

The *synchronized* modifier is used to implement mutual exclusion in Java. Each Java class-instance has a lock [Gosling et al. 1996]. A thread that invokes an instance's synchronized method implicitly requests the instance's lock. If the lock is owned by another

```
class buffer
{
   List list;

   synchronized Object get()
   {
      if (list.isEmpty())
         try {wait();}
          catch (InterruptedException ex){};
      return list.get();
   }

   synchronized void put(Object obj)
   {
      list.put(obj);
      notify();
   }
};
```

Figure 3.1. Java Wait-Notify Interaction

thread, the requesting thread is blocked until the lock becomes available to it. The MPS uses synchronized methods to synchronize MpsMessage-acquisition and transfer.

The *wait* and *notify* methods are used to implement condition synchronization in Java. A thread that calls a instance's wait-method will block and be added to the instance's *wait-set* until another thread invokes the instance's notify-method [Gosling et al. 1996]. *Notify* will select one of the waiting threads, remove it from the wait-set, and make it ready to run. The selected thread will throw *InterruptedException* when it runs again.

The sample code in Fig. 3.1 depicts a basic Java *wait-notify* interaction in a buffer implementation. A consumer-thread that calls the *get*-method shown in Fig. 3.1 will block at the wait statement if the list is empty. The list will be non-empty after a producer-thread calls *put*. The put-operation calls *notify*, causing InterruptedException to be thrown to one of the consumer-threads waiting for the buffer's lock. The interrupted consumer will exit the *catch* clause after the producer-thread releases the lock. The consumer-thread will make a transition from the ready state to the running state.

a. Activator Creation and Start-up

Role-specific behavior is acquired by a thread by overriding the thread's predefined *run*-method. A thread's run-method may execute only after its *start*-method has been invoked.

An SMPS activator-instance is created in the *Created* state when a thread invokes the activator's constructor. Figures A-4 and 3.2 show the TH1-transition from *Created* to *Starting and* Fig. A-4 shows the SM-action for the *Starting* state. In *Starting*, the activator sends TH1 to the activator-thread's state-model causing a transition to the *Ready* state. The RTE makes the thread the currently running thread by sending TH2 to its state-model. This simulates Java *start*-method execution and causes the transition from the *Starting* state to the *Getting Message* state shown in Fig. 3.2 and Fig. A-4.

The SMPS model of an MPS activator primarily models its run-method. The activator run-method repeatedly acquires a message from its message-buffer, updates its EventClock-instance, executes the message activate-method, and transfers the generated messages from temporary storage to their target buffers. Figure 3.3 shows an abridged version of the source code for the MPS activator run-method.

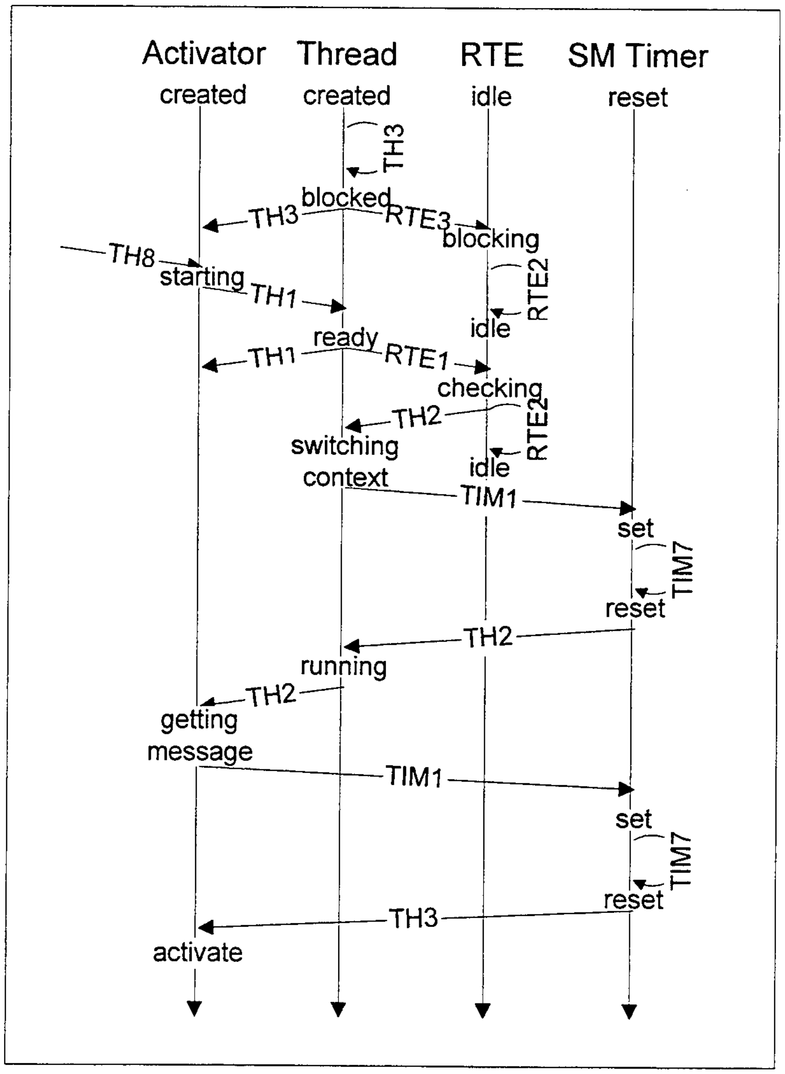

Figure 3.2.
State-Event Chart for Message Acquisition with an Initially Non-Empty Message-Buffer

b. Non-Blocking Message-Acquisition

Line four in Fig. 3.3 shows an example of MpsMessage-acquisition. Figure 3.2 depicts MpsMessage-acquisition in a scenario with an initially non-empty message-buffer. An MPS activator that invokes *get* for a non-empty message-buffer will not block. This is modelled by the *else* branch in the *Getting Message* action shown in Fig. A-4.

An SMPS activator must compute the CPU-time consumed by the Event-Clock update operation and the operation that removes the message at the front of its message-buffer. It must also schedule itself to complete its message-acquisition after that amount of time has elapsed. This is shown by the transition labelled TH3 that goes from the *Getting Message* state to the *Activate* state shown in Fig. A-4.

Message-acquisition CPU-time consumption is simulated by sending TIM1 to an SMM timer-instance. Timer expiry is represented by the second occurrence of TIM7 in Fig. 3.5. The timer sends TH3 to the activator when the correct amount of CPU-time elapses.

Priority inversion is the condition in which a higher priority thread is

```
1  public void run() {
2     setPriority(MPS.MAX_PRIORITY);
3     do {
4        MpsMessage msg = tbuff[id].get();
5        if (msg == null) {
6           break;
7        }
8        eclock.update();
9        setPriority(pri);
10       msg.activate();
11       setPriority(MPS.MAX_PRIORITY);
12       transfer();
13    } while (forever);
14 }
```

Figure 3.3. MPS Activator Run-Method

blocked by a lower priority thread. Consider a Java program with three threads T1, T2, and T3 listed in ascending priority. Assume that they use the buffer in Fig. 3.1 and that it is initially non-empty. The following sequence of events will cause priority inversion because T1 will be blocked by T2's execution.

- T1 invokes buffer.get and acquires the lock
- T2 and T3 become ready to run when T1 is at line seven in Fig. 3.1
- T3 preempts T1
- T3 invokes buffer.put, requests the lock, and is added to buffer's wait set
- T2 runs

Activators raise their priority to the maximum in the *Starting* state or the *Transferring Msgs* state to prevent preemption during message-acquisition and transfer. This also avoids activator-buffer and delay-queue lock-contention and precludes the necessity to model lock-contention in the SMPS. It also avoids blocking-delay due to priority inversion.

c. *Blocking and Unblocking in Message-acquisition*

Figure 3.4 depicts a scenario with two activators, A1 and A2. The scenario begins at the point in which A2 begins to execute the SM-action of the *Getting Message* state. The empty message-buffer condition will cause A2 to execute the *if*-branch in the *Getting Message* SM-action shown in Fig. A-4 and to send TH4 to itself. TH4 will cause it to transition to the *Waiting for Message* state depicted in Fig. 3.4 and Fig. A-4. The self-directed TH4 event simulates the effect of executing a wait-method such as the one depicted on line seven of Fig. 3.1.

The following sequence of events models the process by which an activator may become blocked at a wait statement like the one depicted at line eight in Fig. 3.1. In the W*aiting for Message* SM-action, A2 sends TH4 to its thread state-model, causing the thread to enter the *Blocked* state. In the *Blocked* SM-action, the thread sends TH3 to A2 and

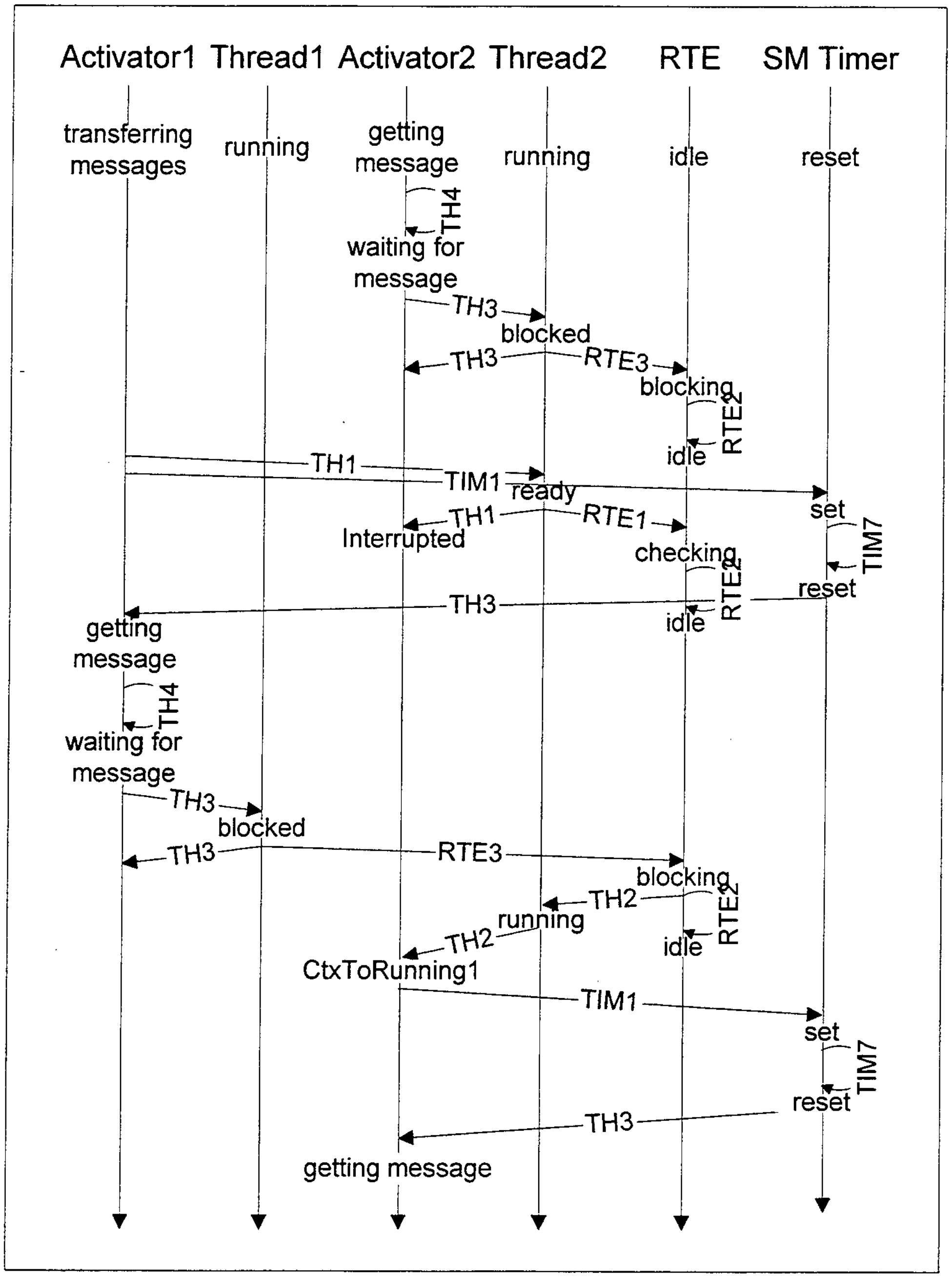

Figure 3.4.
State-Event Chart for Message Acquisition with an Initially Empty Message-Buffer

sends RTE3 to the RTE. RTE3 causes the RTE to enter the *Blocking* state and it may execute a context switch.

After A2 has become blocked, A1 sends TH1 to A2's thread from the *Transferring Msgs* SM-action. This models the effect of invoking *notify* in a *put* operation as shown at line 16 of Fig. 3.1. TH1 causes A2's thread to transition from *Blocked* to *Ready* and to send TH1 to A2, which is ignored. When the RTE moves A2's thread from *Ready* to *Running*, the thread sends TH2 to A2, causing it to go from *Waiting for Message* to *Getting Message*. A2 continues from that point as described in the discussion for non-blocking message-acquisition.

d. Activate-Method Execution and Preemption

Line eight in Fig. 3.3 depicts an activator that acquires a message and updates its EventClock-instance to the current system time. The activator reduces its priority after it updates the EventClock and then invokes its activate-method. The activate-method will only modify the values of attributes in the same AO_p and it may place generated-events in temporary storage. However, no simulation-time will elapse during activate-method execution in the SMPS.

Activate-methods must be invoked at the same time in the SMPS as they would have been in the MPS in order to be AIO-consistent. Activate-method completion-time is the time at which an activate-method completes in the MPS. The SMPS models this as the sum of activate-method start-time plus the CPU-time consumed by the activate-method plus its preemption-time plus the context-switching time associated with preemption.

The activator must delay the occurrence of two operations until activate-method completion-time to achieve AIO-consistency. First, it must delay the transfer of the messages from temporary storage to message-buffers. Second, it must delay the start of the next activate-method.

The activator-class defines an *add2ActionTime*-method that adds an increment of simulation-time to the value of an internal action-time variable. In the SMPS, this variable's value is reset prior to each activate-method invocation.

The activate-method uses *add2ActionTime* to dynamically accumulate activate-method timing information during execution. The action-time variable should contain the amount of CPU-time that would be consumed during MPS-based activate-method execution in the target environment. Action-time is unused in the MPS. However; it is used by the SMPS to compute activate-method completion-time.

An activator executes the activate-method in the SM-action of the *Activate* state shown in Fig. A-4. The activator transitions to *Activating* after completing the activate-method. The transitions between *Activating* and *Preempted* simulate the passage of activate-method execution-time and preemption-time.

Preemption can only occur in the MPS during activate-method execution because message-acquisition and message-transfer are performed at elevated priorities. Preemption only occurs when a timer's timeout expires and the timer transfers a message to an activator-buffer owned by a waiting activator. The message-transfer unblocks the waiting activator as described in the discussion on blocking message-acquisition. The activator will also continue processing until it reaches the *Activating* state. At this point, the activator will have executed its activate-method without affecting any objects in other

AO_ps due to AE-predictability constraints. If the activator lowers it priority to a lower priority than the preempted activator's priority, the RTE will perform a context-switch and restore the preempted activator to the running state.

The amount of time elapsed during preemption will be equal to the sum of the timer message-transfer time and the activator message-acquisition time. Although its activate-method will have executed, activate-method execution-time does not affect the simulation clock at that point.

The state-event chart shown in Fig. 3.5 shows the sequence of events and state transitions involved in preempting activator A1 in the *Activating* state. Initially, the SMPS timer will have scheduled an event-delivery by setting the SMM timer to expire at a later time. This timer expiry is depicted by the appearance of TIM7 in the upper part of the SMM timeline in Fig. 3.5. The SMM timer sends TH1 to the SMPS timer-thread when the SMM timer expires. The SMPS timer-thread interrupts the SMPS timer and the SMPS timer proceeds through a succession of state transitions causing it to send RTE1 to the RTE. The RTE compares A1's priority and the timer's priority and sends TH1 to the A1-thread and TH2 to timer-thread because the timer-thread has a higher priority than A1. The A1-thread transitions to the *Ready* state and sends TH1 to A1. A1 transitions to the *Preempted* state when it receives TH1.

The simulation clock is updated twice during this series of interactions. First, it is updated at the time of timer expiry depicted by the appearance of TIM7 in the upper part of the SMM timeline shown in Fig. 3.5. It is updated again at the second appearance of TIM7 that represents the context-switching time of the timer-thread when it transitions from *CtxToRunning1* to *Getting Message*. A1 computes *Tremain,* the amount of activate-method execution time remaining, by subtracting its last activation start-time from the current simulation-time. When the RTE causes it to transition to the *Activating* state again, it will recompute its activate-method completion-time as the sum of the then-current simulation-time plus *Tremain*.

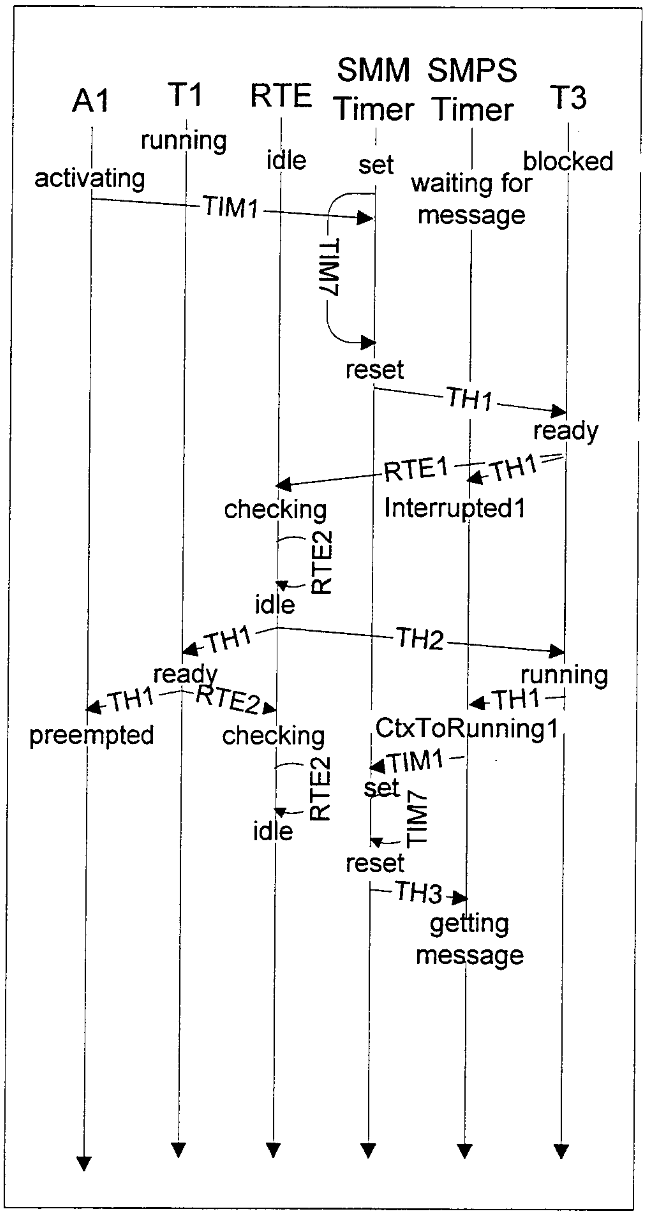

Figure 3.5. State-Event Chart for Preemption

3. The Timer

Activators remove the message from the buffer in the *Getting Message* state and transfer messages from the buffer in the *Transfer Msg* state. However, the timer does not remove the message from the delay-queue until it reaches the *Transfer Msg* state because a message with an earlier transfer time can be inserted into the delay queue while the timer is in the *Timeout* state. The timer is never preempted because it always executes at the thread message-transfer and message-acquisition priority.

The delay-queue maintains a list of ScheduledMessage-instances that are sorted by activation-time. Figure A-3 shows that the timer transitions to the *Timeout* state when it identifies the message at the front of the delay-queue. In the *Timeout* SM-action, the timer sends TH3 to its thread and schedules TH1 to be sent to its thread after the timeout period elapses. These events cause the timer-thread to transition immediately to *Blocked* and later to *Ready* when the timeout elapses. Figure 3.5 shows that the SMM timer sends TH1 to the SMPS timer-thread. That causes the SMPS timer-thread to transition to the *Ready* state. The timer-thread sends TH1 to the timer, which is ignored.

The delay-queue put-method compares the activation-time of the new MpsMessage argument against the timer's current-message activation-time. If the new message activation-time is earlier, then it pushes the current-message and the new message argument onto the delay-queue. It then sets the current-message to null and sends TH1 to the timer's thread. This causes the timer to process the earlier message.

D. SUMMARY

The application-binary-code consistency (ABCC) requirement is that the same application binary-code execute in both the target execution-environment and in the simulation model. The investigation used two implementations of a message-passing architecture, the MPS and the SMPS, that implement state-machine-execution engines to support application-binary-code consistency (ABCC).

Several MPS design-properties contribute to ABCC. First, each SM-action execution was implemented with a single message-activation. In the SMPS, each SM-action execution was implemented with one DES-action. Second, the MPS and SMPS message-activation methods must generate the same final state, given the same initial state. Third, an MPS-based program must execute its SM-action methods at the same time in the SMPS as in the MPS. Finally, data access and synchronization policies must assure mutually exclusive data access.

An MPS-based application can be ABCC for an MPS and SMPS that have these design properties if the variations in its sub-path execution-times only results from preemption by another MPS thread. The SMPS sub-path execution times are obtained from an MPS platform performance model (MPPM). However, if execution conditions are impossible to specify and control, then the MPPM will be invalid and ABCC cannot be achieved.

IV. MPS PLATFORM PERFORMANCE MODELLING

The first section of this chapter discusses MPS platform performance model (MPPM) requirements for validity. The second section provides an overview of dual-loop (DL) testing, and describes DL test structure principles, design considerations, and side effects in DL testing. The third section describes MPPM development in the investigation used in this thesis. The fourth section describes how the MPPM was incorporated into the SMPS. The fifth section provides conclusions on MPS platform performance modelling.

A. MPPM REQUIREMENTS

The validity of the SMPS and its associated MPS-based application depends partially on the fidelity of the MPPM. An MPS platform performance model (MPPM) is a set of functions that return MPS execution-time estimates (ETE) that model the execution-times of MPS module sub-paths under MPS-application execution conditions. A Java compilation configuration comprises the Java compiler and the compiler options used to compile the MPS and MPS-applications. An MPS execution platform consists of a Java virtual machine (JVM), an operating system, and a hardware platform. MPS-application execution conditions are defined by a Java compilation configuration, an MPS execution platform configuration, and a system loading specification for a particular MPS-application execution.

An SMPS execution has strict MPPM-validity if the MPPM functions return MPS ETEs that are the same as the corresponding MPS sub-path execution-times at the same time relative to the program execution start time. This definition does not require or assume PET-consistency. An SMPS execution that has strict MPPM-validity is AIO-consistent with an MPS program execution and can be used to reproduce MPS application behavior in a run-time environment that is free of the probe effect.

The MPS execution platform used for the investigation presented in this thesis was a 200 MHz Pentium PRO computer with a 256 KB on-chip cache and 96 MB of 60 ns RAM running under Windows NT 4.0 and the Java Development Kit (JDK) version 1.1.6. If MPS execution conditions are impossible to specify and control for such a configuration then a strictly valid MPPM cannot be developed. In addition, MPS applications executing on an unpredictable platform are not PET-consistent. This raises the question of whether an MPPM can be developed that is sufficiently predictable to support useful software testing in contrast to providing strictly reproducible behavior. An MPPM is useful if the sum of the effort devoted to its development and to testing a set of control-flow paths in the SMPS is less than the effort required to test the same paths in the MPS. MPPM usefulness is related to the difficulty of testing the real system.

A partial MPPM was developed for this configuration that consists primarily of Java language feature ETEs. This MPPM contains of a set of mappings of feature-names to ETEs and does not account for performance variations due to system loading or properties of the execution platform.

B. DUAL-LOOP (DL) TESTING OVERVIEW

1. DL Test Structure

The DL approach is based on the assumption (i.e., the DL-assumption) that the difference in execution-times of two segments of code that are identical except for the presence of a feature-of-interest is equal to the execution-time of the feature-of-interest. A dual-loop test generates an ETE of a feature-of-interest by comparing the execution-times of an experimental loop and a control loop. The two loops are identical except that the experimental loop contains the feature-of-interest. For control loop execution-time, T_c, and experimental loop execution-time, T_e, the feature-of-interest execution-time, T_f, is computed as $T_f = T_e - T_c$ [Pollack and Campbell, 1990].

Let M_{el} and M_{cl} be the measured execution-times of N iterations of the experimental loop and N iterations of the control loop. Let M_f be the measured feature execution-time computed as $M_f = (M_{el} - M_{cl})/\mathrm{N}$. The basic clock resolution of the clock-function, τ, is usually too coarse to permit M_{el} and M_{cl} to be used as estimates of T_e and T_c when N=1. The time measurement uncertainty is τ for M_{el} and for M_{cl}, so $NM_f - 2\tau \leq NT_f \leq NM_f + 2\tau$. The value of N required to confine the measurement uncertainty to a fraction, p, of NM_f, is given by $N \geq 2\tau/(pM_f)$ [Pollack and Campbell, 1990].

Altman examined the DL-assumption after measuring negative ETEs for some DL benchmarks written in Ada [Altman, 1987]. He attributed some of the main sources of distortion in test results as being due to design or implementation errors and interference from side-effects. Design and implementation errors can be addressed. However side-effects are due to external factors that can be impractical or impossible to precisely quantify and control.

2. DL Test Design Considerations

An in-depth knowledge of the target programming language may be required to avoid DL test design errors and to avoid compromising MPPM fidelity. An MPS application that uses many language features that differ in subtle ways can necessitate a large test suite to generate an MPPM for an MPS application. For example, consider the problem of measuring Java calling overhead for methods with various numbers of arguments and various argument types. A conservative assumption is that any different signature type has potentially different performance implications.

Table 1 shows an example of variability in Java method signature types based on the conservative assumption. The top row contains the variability category for each column. The last row contains the number of values for each category. The intermediate rows display the names of the values in the category or an indication of the presence or absence of a value in the category. Some of the factors pertain to the value or presence of a *throws* clause, a *synchronized* modifier, a *static* modifier, *access* modifier, or a *final* modifier. The others pertain to the method return type and the argument types.

The number of unordered combinations of argument types that can be constructed from three arguments and seven argument types is $(k+n-1)!/(n-1)!k!$ where n is one more than the number of data types and k is the number of arguments. This is analogous to determining the number of different colorings of k golf balls with n colors [Anderson, 1974]. The extra argument corresponds to the *void* argument or the absence of an argument. The number of different unordered combinations is 120 (i.e., (8+3-1)! / (8-1)!*3!). The number of different signatures is 30720. This value is the product of the number of factors in each category multiplied by the number of different combinations of arguments (i.e., 2*2*2*2*4*8*120).

Not every kind of method signature variation affects performance. Some of the Java method modifiers probably only affect compile-time consistency checks. Table 2 shows an optimistic interpretation of variability in Java method signature types. The categories in table 2 are partially based on properties in generated code that are assumed to affect performance. For example, methods that have the static, final, or private modifiers or that are declared in a class that has the final or private modifiers may be able to be inlined. The table also assumes that argument size rather than argument type is relevant to method-calling overhead execution-time.

throws	synch. mod.	static mod.	final mod.	access mod.	return type	argument type
present	present	present	present	*public*	*void*	*char*
absent	absent	absent	absent	*pro-tected*	*char*	*boolean*
				private	*boolean*	*int*
				absent	*int*	*long*
					long	*float*
					float	*double*
					double	"refer-ence"
					refer-ence	absent
2	2	2	2	4	8	7

Table 4.1: Conservative Interpretation of Variability in Java Method Signature Types

The number of argument signature types that can be constructed from three arguments and four data types is 35 (i.e., (5+3-1)! / (5-1)!*3!). The number of different signature types based on the optimistic assumption is 140.

synch. mod.	inline-able	argument size (bytes)
present	true	1
absent	false	2
		4
2	2	4

Table 4.2: Optimistic Interpretation of Variability in Java Method Signature Types

synch. mod.	inline-able	argument size (bytes)
		8
2	2	4

Table 4.2: Optimistic Interpretation of Variability in Java Method Signature Types

3. Side Effects

The MPS compilation and execution environment contains several possible sources of instability for MPS sub-path execution-time. These sources of instability can be attributed to the JDK 1.1.6 JVM, Windows NT 4.0, or the Pentium PRO hardware platform. The JDK 1.1.6 JVM contains a bytecode interpreter and a Just-in-time (JIT) compiler. The JIT performs adaptive optimization at run-time by determining the methods that are called most often and by generating machine code for them on-the-fly [Yellin, 1996]. This can lead to variations in sub-path execution-times in the absence of other factors. The JVM also performs garbage collection at unpredictable times.

Windows NT 4.0 is a preemptive multi-tasking operating system that performs various operating system services in independent execution threads. These can preempt an MPS application and result in variations in sub-path execution-time. The Pentium PRO processor has a high-speed memory cache. Variations in the sequence of memory fetches can affect the percentage of cache hits and lead to a differences in sub-path execution-times.

C. MPPM DEVELOPMENT

1. The MPPM of the Passage of Time

It is essential to obtain an estimate of τ, T_{cr}, because τ is integral to the application-level view of the passage of time. At any time, T, the value returned by the Java clock function, *system.currentTimeMillis*, will be $\lfloor\lfloor(T/\tau)\rfloor\tau\rfloor$, where $\lfloor\ \rfloor$ is the *floor* function. The value of τ is also required to compute the number of iterations to execute the control and experimental loops that are used measure the execution-time of a particular code segment. The value returned from *system.currentTimeMillis* was assigned to each element of a large array. The sample values consisting of the non-zero differences between successive array elements were counted and summed to obtain T_{cr}. This process was repeated ten times and the average of the samples, T_{cr}, was computed to be approximately 15.625 milliseconds (ms). The source code for the test program is shown in Fig. B-1.

Each MPS and SMPS *activator* updates its event-clock with the current value of the *SystemClock* at the beginning of each action as shown in Fig. A-3 and Fig. A-4. The *SystemClock* function was implemented in the MPS by calling the Java *System.currentTimeMillis* method from the *MpsRealClock.millis* method. In the SMPS, the *MpsRealClock.millis* method models the Java *System.currentTimeMillis* method. This provides a more accurate application-level view of current-time than simply returning the value of the EDC-clock. The SMPS *MpsRealClock.millis* source code is shown in Fig. B-2.

Figure B-3 shows the source code for the test to determine the Java *wait-method* resolution. The test performed 30 invocations of the *wait*-method for each value in the range of one to 50 milliseconds. Figure 4.1 shows the results of the test. Requested delay is shown on the abscissa and measured delay is shown on the ordinate. In each case, the measured delay resolution is T_{cr}, but the average measured delay is close to the requested value. The results do not validate a particular model of *wait*-method resolution. However, they do not contradict the optimistic assumption that *wait*-method resolution is high relative to the clock function resolution.The SMPS *timer* implementation design is based on the optimis-

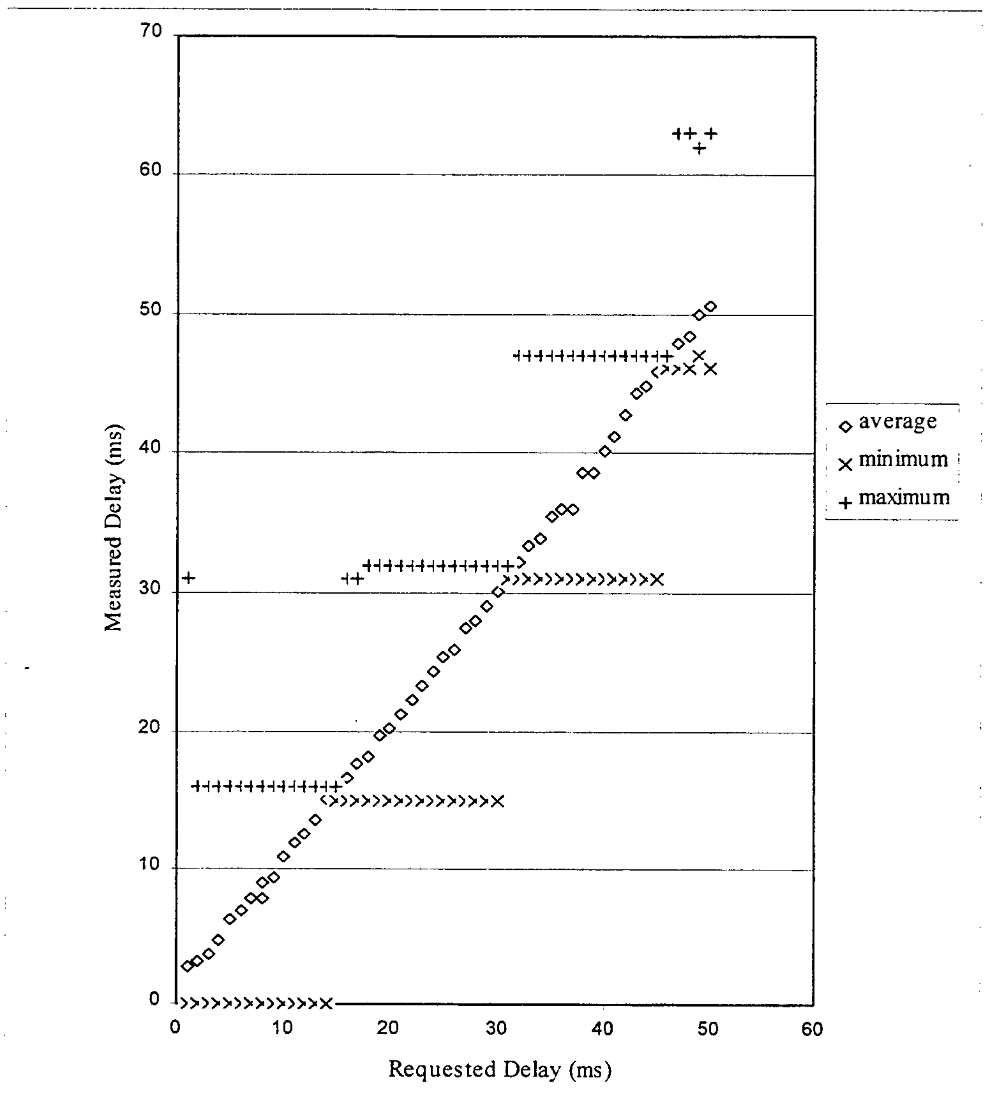

Figure 4.1. Java Wait Statement Resolution for a 200 MHz Pentium Pro with 96 MB 60 ns DRAM and 256K On-Chip Cache

tic assumption and uses the EDC-clock rather than the SMPS *MpsRealClock* to compute delay-times for *delayed-event-delivery*. However, the application-view of *delayed-event-delivery* duration can only be accessed using *MpsRealClock.millis* so its view of the clock function resolution is still T_{cr}.

D. GENERAL DL TEST DESIGN AND IMPLEMENTATION

The majority of the DL tests were implemented as type extensions of the abstract

FeatureTest class. *FeatureTest* declares the following four abstract methods: *initialize, calibrate*, and *execute*. Each concrete subclass must implement those three methods. *Initialize* initializes the test state. *Calibrate* executes multiple iterations of a loop that exercises the control and experimental loops *N* times per iteration, and doubles *N* for each successive iteration. It calculates T_f for a feature and completes when $T_f \geq (2T_{cr})/p$, where p=0.01. *Execute* re-computes T_f at the calibrated value for *N*, *M* times, where *M*, is the number of test repetitions. Initially, *M* was set at 29, so a total of 30 repetitions of each test was performed. However, *M* was eventually reset to zero because there was so little variation between successive runs.

Figure 4.2 shows the test results and output of a method calling overhead test, program *smsipubv3pMf*. Program *smsipubv3pMf* measures the calling-overhead of a method with the *public, static*, and *synchronized* modifiers. The method also has three integer arguments and no return value. Figure 4.2 shows that *N* ranged from 2^{16} to 2^{21} during the calibration portion of the test. The variation in M_f was less than 0.4% of the mean for the five runs and the average value for M_f was approximately 2.12 microseconds.

E. INCORPORATING THE MPPM INTO THE SMPS

The *CpuCharger* interface requires an implementation to provide methods that add a quantity of time to an accumulator, clear an accumulator, and return the accumulated value. The SMPS *activator* and *timer* implement the *CpuCharger* interface and use *CpuCharger* methods to maintain a record of estimated CPU-time use. The *activator* defines an *add2ActionTime*-method that adds an increment of simulation-time to the value of an internal action-time variable. This is exported to MPS-applications and provides a way for MPS applications to communicate activate-method CPU-time consumption to *activators*.

The ETEs obtained from the DL tests were stored as a set of mappings of DL test-names to ETEs. Java statements and ETE names were embedded in special comments in the SMPS source code. Figure B-4 shows the SMPS *MpsFifo.append* method with embedded Java statements and ETE-names. The source was submitted to a processor that strips out the special comments and replaces the ETE-names with their values. Figure B-5 shows the result of processing *MpsFifo.append.*

Figure B-5 shows that *Vappend_1mf* contains the sum of the ETEs for the *append*-method calling overhead, the *if*-statement evaluation, and the statements in the first branch of the *if*-statement. *Vappend_2mf* contains the sum of the ETEs for the *append*-method calling overhead, the *if*-statement evaluation, and the statements in the second branch of the *if*-statement. Each branch invokes *cc.charge* which adds the DD-path ETE to the CPU-time

```
----- Feature Time Estimates -----
executing method.smsipubv3pMf test
N:65536, Tf:141, Te:172, Tc:31
N:131072, Tf:297, Te:344, Tc:47
N:262144, Tf:563, Te:672, Tc:109
N:524288, Tf:1124, Te:1359, Tc:235
N:1048576, Tf:2235, Te:2688, Tc:453
N:2097152, Tf:4453, Te:5359, Tc:906
method.smsipubv3pMf
Series 0, Number of tests 5, N: 2097152,
p: 0.01, units: ms
+
Test, Te, Tc, Tf, Mf
1, 5359, 906, 4453, 0.002123355865478515 6
2, 5328, 891, 4437, 0.0021157264709472656
3, 5328, 891, 4437, 0.0021157264709472656
4, 5328, 890, 4438, 0.0021162033081054688
5, 5344, 891, 4453, 0.0021233558654785156
+
Mean: 5337.4, 893.8, 4443.6, 0.0021188735961914064
--- registered ---
```

Figure 4.2. Output from the *smsipubv3pMf* Dual-loop Test

accumulator for the SMPS thread.

Language features such as conditional evaluation and self-modifying expressions should be avoided unless the DD-path execution can be known ahead of time, since they contain more than one DD-path in a single statement.

F. CONCLUSIONS

A partial MPPM was developed for the target platform using the dual-loop approach to measuring execution-time. The validity of an MPPM developed using the DL test approach on a predictable platform is related to the validity of the DL assumption. A large number of DL tests may be required to accurately reflect DD-path execution-times due to programming language subtleties. Good DL test design requires an in-depth knowledge of the target programming language.

Factors such as the JDK 1.1.6 just-in-time compiler, the Windows NT 4.0 preemptive multi-tasking operating system, and the Pentium PRO high-speed cache are sources of unpredictably in the MPS execution platform. If MPS execution platform performance is unpredictable then MPS-application behavior cannot be precisely reproduced using the SMPS. However, an MPPM can be useful if its reduces the effort to test control-paths that are difficult to reproduce in real systems.

V. DEMONSTRATION PROGRAMS

This chapter is concerned with demonstrating and evaluating the synchronization and timing properties of MPS-based applications running with the SMPS. Two MPS-based programs were developed that illustrate well-understood synchronization and timing principles. The first is an MPS-based solution to the “Dining Philosophers” problem [Dijkstra, 1971] that was developed to study SMPS timing and synchronization. The second is an MPS-based program with Rate Monotonic Scheduling (RMS) priority assignments and artificial workloads [Sha and Goodenough, 1990]. This program was developed to study SMPS timing and the model of a preemptive multi-threaded run-time system.

The first section describes the “Dining Philosophers” problem and the MPS-based solution implemented with the uniprocessor-based SMPS. It also compares timing and synchronization in the actual program to timing and synchronization in the ideal case. The second section describes the RMS program and compares timing and preemption in the actual program to timing and preemption in the ideal case. The third section presents the conclusions and describes the implications for the distributed systems question.

A. DINING PHILOSOPHERS (DP)

In this problem, five philosophers are seated at a round table. Each philosopher has a plate, and between each pair of plates is a chopstick. A philosopher must acquire the chopsticks to the immediate left and right before he can eat. This prevents adjacent philosophers from eating at the same time. Each philosopher repeats a cycle of acquiring chopsticks, eating, relinquishing chopsticks, and then thinking, until no more rice remains. If each philosopher initially picks up the chopstick on the left before any philosopher has a chance to pick up the one on the right, deadlock will occur. Deadlock will not occur if a server assures that no more than N-1 philosophers are allowed to eat at a time [Burns and Wellings, 1995].

The test scenario used a time-scale measured in milliseconds to allow target platform overhead effects to be expressed in the actual DP-program execution. DP eating and thinking activity-durations were chosen so that the ideal case would exhibit both concurrent and non-concurrent eating periods over a relatively small time period.

The initial condition for the test scenario consisted of five philosophers and ten available bites of rice. Each eating-period duration for the philosophers Plato, Hegel and Descartes, was 150 milliseconds (ms) and their thinking-period duration was 250 ms. The eating-period duration for Lao Tsu and Socrates was 250 ms and their thinking-period duration was 500 ms. Figure 5-1 shows the physical arrangement of the plates and chopsticks for the five philosophers seated at the table. The OOA models shown in Fig. C-1 through

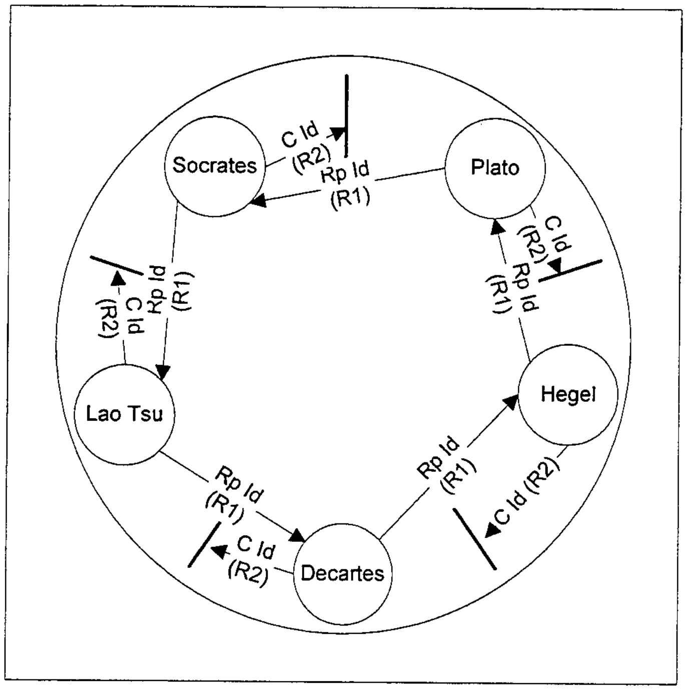

Figure 5.1. Physical Arrangement of Dining Philosophers

Fig. C-6 specify the solution implemented in the MPS-based DP test application. The arrows in Fig. 5-1 represent relationships R1 and R2 in the DP object information model (OIM) shown in Fig. C-1. The arrows emanate from the seating position for the philosopher that owns the referential attributes for the designated relationship. Relationship R1 denotes the philosopher seated to the right of the philosopher that owns the referential attribute that formalizes relationship R1. Relationship R2 denotes the chopstick to the left of the philosopher that owns the referential attribute that formalizes relationship R2.

The ideal case was defined as hypothetical DP-program execution on a platform with a perfect system clock an no execution overhead. The ideal case is shown in Fig. 5-2. Thinking and eating activities are indicated for each philosopher by relatively lower and higher line segment height. The ten line segments with the highest of the three heights identify the times that the rice was consumed. Plato and Descartes eat simultaneously between 1350 ms and 1400 ms and Plato and Lao Tsu eat simultaneously between 1400 ms and 1500

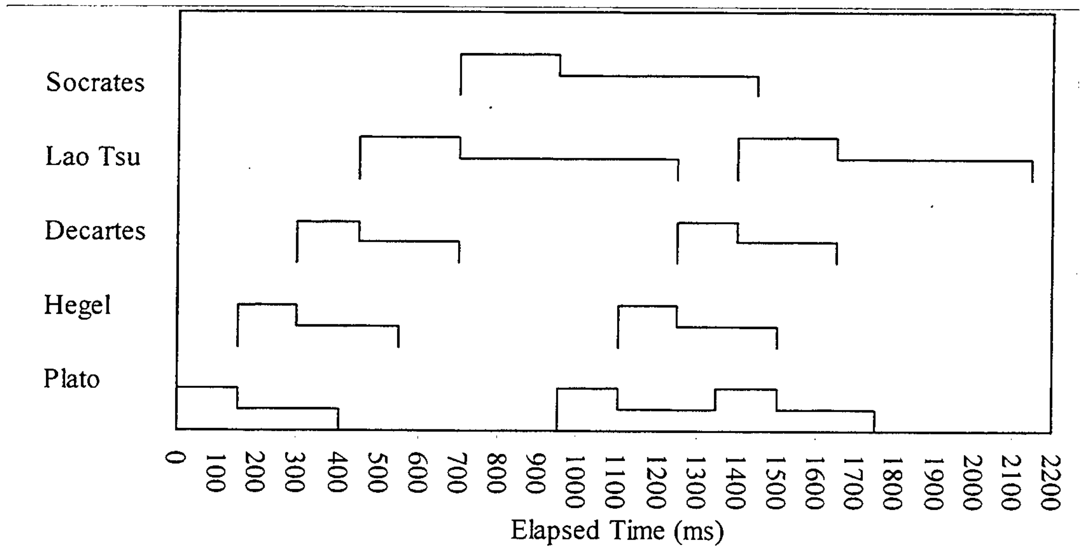

Figure 5.2. Dining Philosophers: Ideal Case

ms. Figure 5-1 shows that Plato is not adjacent to Lao Tsu or Descartes.

No satisfactory estimate of context-switch time was obtained during the investigation presented in this thesis. Context-switch time was measured as the time taken for a number of threads, *N*, that were blocked at a *wait* statement to run. This approach was not perfected. Available memory also limited *N* to 1200, and *N* was not large enough to meet the accuracy criterion of $N \geq 2\tau/(pM_f)$ for p=0.1. Nevertheless, the context-switch time estimate of 0.7 ms was incorporated into the MPPM to demonstrate SMPS functionality.

The other feature-time estimates (FTE) are relatively precise under the DL assumption. The number of control loop and experimental loop iterations, *N*, ranged from 2^{19} to 2^{27}. The measurement error for NM_f was $\pm 2\tau$, so the accuracy of M_f was computed as $2\tau/N$. This yielded accuracies that ranged from ±3x10-7 ms to ±6x10-5 ms for FTEs that were obtained with the standard DL approach. This demonstrates the potential to acquire FTEs

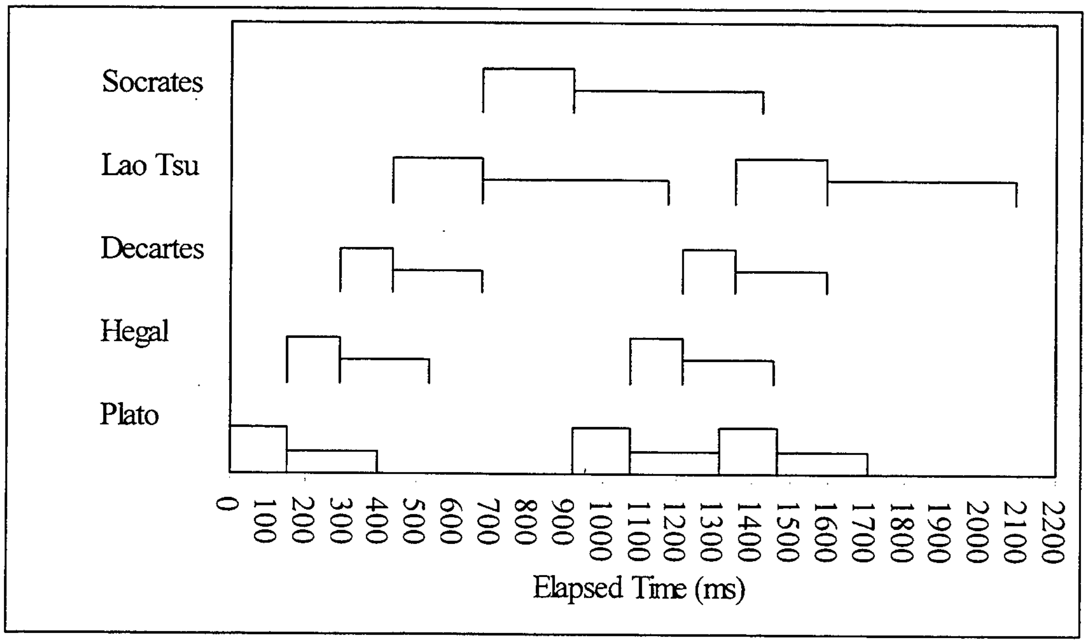

Figure 5.3. Dining Philosophers: SMPS Execution

that are relatively accurate under the DL assumption.

The actual SMPS DP-program execution used the partial MPPM described in chapter four. Figure 5-3 shows that the eating and thinking order is the same in both the SMPS DP-program and the ideal case. However, there are relatively small differences in timing between the two. These differences are due to differences in the models of system time and the contribution of the MPPM to SMPS timing. In the ideal case, no time elapses during Plato's transition from *Created* to *Wait* shown in Fig. C-2. However, 1.4017 ms elapses during the same transition in the SMPS DP-program. This is shown in section D.1 of Appendix D, which contains the application-level program-traces for the SMPS DP-program. Lines two through seven in section D.1 indicate that 1.4017 ms is consumed by Plato's transition from *Created* to *Wait.* Section D.2 shows the SMPS-level program traces that correspond to lines six and seven in section D.1 This shows three components to the transition time. The first is the 0.7 ms start-up context-switch time for the timer thread indicated on lines 14-15. The second is the 0.7 ms start-up context-switch time for the activator thread shown on lines 23-25. The third is the activator's 0.0017 ms message acquisition-time for the initial *P1* event for the transition from *Created* to *Wait* that is indicated on lines 23-25.

In the ideal case, Hegel begins to eat a bite of rice during 150-300 ms. However, line 59 in section D.1 shows that at 152.9148 ms, Hegel schedules eating to complete at T=290 ms. The actual completion time is shown on line 62 to be at T=291.4098 ms. The difference between the ideal and actual start times of T=150 ms and T=152.9148 ms is due to the SMPS model of MPS overhead. However, the difference between the ideal 300 ms completion time and the scheduled 290 ms completion time is due to the model of the application-level view of system-clock resolution. At any time, *T*, the value returned to the application by the simulated system-clock function would be $\lfloor\lfloor(T/\tau)\rfloor\tau\rfloor$. When T=152.9148 and τ=15.625, the value returned from the simulated system-clock function is 140. When the 150 ms eating-time duration is added, the 290 ms time correctly reflects what the completion time should be based on the MPPM used in the SMPS DP demonstration.

B. PERIODIC HARMONIC TASK SET WITH RMS PRIORITY ASSIGNMENT

This MPS-based RMS program was developed to study SMPS timing and the model of a preemptive multi-threaded run-time system. The program modelled three independent periodic tasks: T0, T1, and T2. Each task had a workload and a period in which to complete its work. The ideal RMS scenario was based on the assumption that there was a perfect system clock, no system overhead, and no MPS overhead.

The workloads (*wl*) chosen for the experiment were 20, 40, and 80 milliseconds and task periods (*pd*) were 62, 124, and 248 milliseconds, respectively. Priority assignments were inversely related to task period, in accordance with RMS principles. These conditions were designed for high CPU utilization so that the addition of MPS overhead in the SMPS version might cause a task to fail to complete its workload by the end of its period.

All three tasks were started at the same time and timing data were gathered until the end of lowest priority task's first period. The RMS critical zone theorem states: "For a set of independent periodic tasks, if each task meets its first deadline when all tasks are started at the same time, then the deadlines will always be met for any combination of start times" [Sha and Goodenough, 1990].

In the SMPS version, activators A0, A1, and A2 provided the execution context for the three tasks the tasks T0, T1, and T2. Task behavior was modelled with a specialization of MpsObject and a specialization of MpsMessage. The MpsObject specialization had *work* and *wait* methods. *Work* modelled task workload execution by executing the MpsObject *addToActionTime* method. *Wait* modelled *delay* by scheduling the MpsMessage instance to be delivered at the end of the period.

Figure 5-4 shows the task execution patterns for the ideal case. T0 executed its workload four times, consuming 80 ms of CPU time. T1 executed its workload two times, also consuming 80 ms of CPU time. T2 executed its workload once, also consuming 80 ms of CPU time. T2 completed its workload at time T=240 ms, close to the end of its period at T=248 ms. T2 started executing its workload at T=60 ms, but was preempted by T0 at T=62 ms and T=186 ms.

Figure 5-5 shows the task execution patterns for the SMPS version. The SMPS version has three activator threads and the timer thread. The activators execute at their *activate* priority (P_A), or else at maximum priority (P=4). They operate at maximum priority whenever they perform any processing other than *activate* method processing. Figure A-4 shows the points in the state model at which the activators raise and lower priority. The two line segment levels in each activator plot in Fig. 5-5 denote *activate-method* processing and higher priority processing. The timer thread always operates at the maximum priority. The spikes in the timer plot denote points at which the timer either gets a message or transfers a message to an activator buffer. T1 was not preempted in the ideal case because the sum of the workloads for T0 and T1 was less than T1's period. However, there is overhead due to four context switches and message handling before time T=60 ms in the SMPS version. The four context switches consume 2.8 ms of CPU time, in accordance with the MPPM. This is enough CPU time to assure that T1 does not complete its workload within T0's period. This is depicted by the spike at T=85.3 ms in A1's plot and represents the small amount of CPU time remaining for A1 to complete T1's workload. Section D.3 provides detailed program traces for the preemption activity that takes place during the interval from T=62 ms to T=63.4 ms. Context switching and message handling consume enough CPU-time to prevent T2 from completing its workload before the end of its period. T2 executes 73.16 ms of its workload by the end of its period at T=248, with 6.84 ms remaining.

C. SUMMARY

An MPS-based solution to the "Dining Philosophers" (DP) and an MPS-based program with Rate Monotonic Scheduling (RMS) priority assignments were developed to test the uniprocessor SMPS timing, synchronization, and the model of preemption. The actual behavior was compared to ideal behavior of the DP and RMS programs. The SMPS DP program had the same ordering of eating, thinking and waiting activities as the ideal case and appeared to have the same synchronization properties as the ideal case. Small differences between ideal DP timing and actual SMPS DP timing were observed. However, the ob-

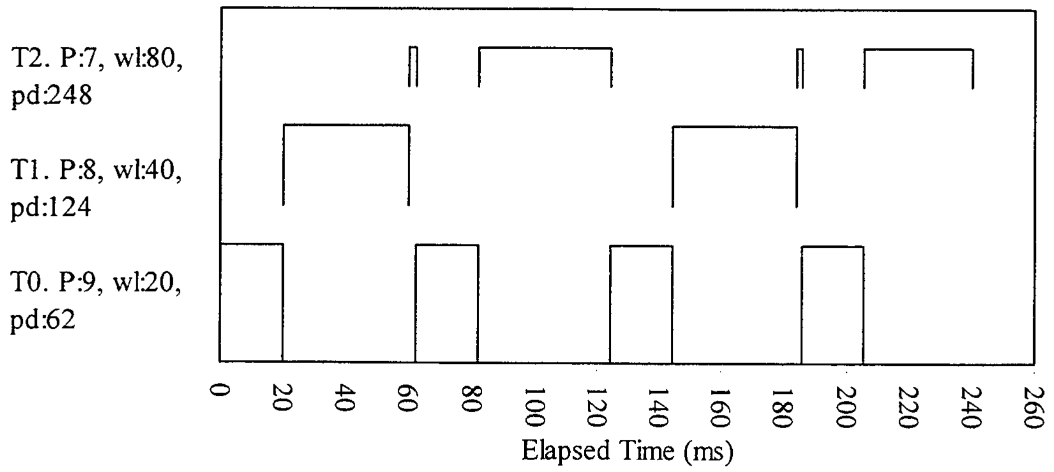

Figure 5.4. Ideal Task Execution Patterns for Three Tasks with RMS Priority Assignments

served differences in timing were found to correctly reflect the MPPM.

The RMS program was designed to exhibit high CPU utilization. The ideal case had eight ms of unused CPU-time in the lowest priority task's first period. In the SMPS version, the lowest priority task failed to complete its workload within its period. This result correctly reflected the effect of MPS overhead for the MPPM used in the experiment.

The results of the demonstrations indicate that it is feasible to develop a discrete-event simulation (DES) implementation of a software architecture that has application binary-code consistency (ABCC) with an implementation designed for a uniprocessor-based target execution-environment. The distributed-systems question adds another dimension to the uniprocessor question, but does not add new research questions. There are two problems in the distributed-systems question. The first is how to model a multiprocessor-based single-platform system. The approach developed in this investigation can address that problem as a variation of the same software architecture. Figure A-7 shows an object infor-

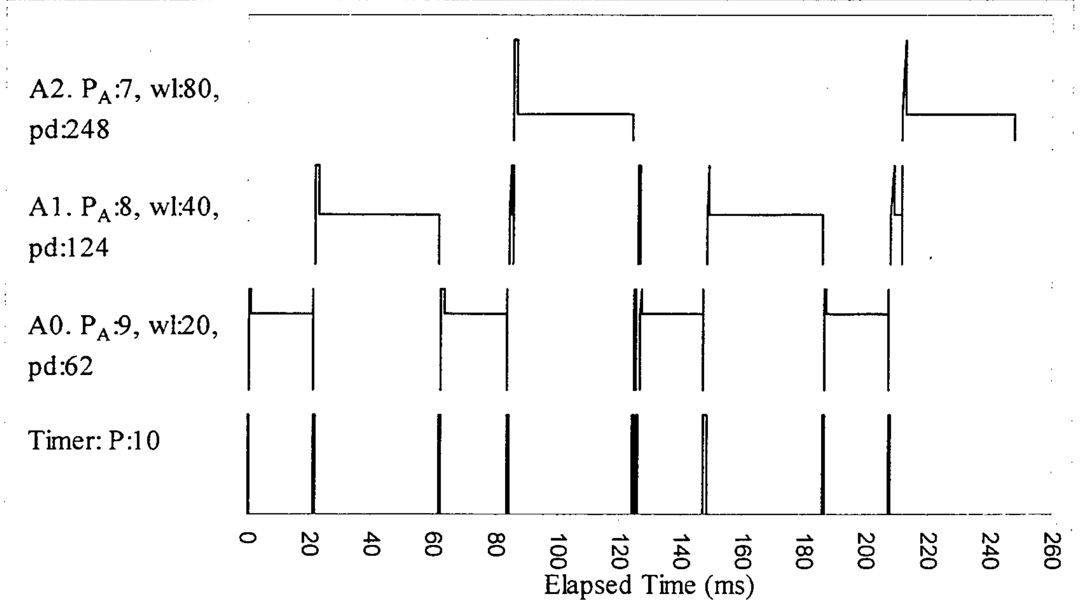

Figure 5.5. SMPS Version of Task Execution Patterns with RMS Priority Assignment

mation model (OIM) for a simulation model of a simulated multiprocessor-based MPS. Relationship R17 shown in Fig. A-7 indicates the possibility of multiple processors under the control of a single run-time executive. Relationship R16 denotes the relationship between threads in the *running* state and the associated CPU. No modifications to the thread state models shown in Fig. A-6 would be required to accommodate a multiprocessor-based model. The RTE state model would be the only state model affected by the new requirement. Figure A-8 shows the state model for a multiprocessor-based RTE. The difference between the SMPS RTE state model shown in Fig. A-5 and the SDMPS RTE state model shown in Fig. A-8 is that the SDMPS RTE assigns eligible threads to processors.

The second problem in the distributed-systems question is how to model multiple communicating platforms. This would require modelling the communication system as a separate architectural domain. Simulation models of communication networks are often used to assist in making design decisions and to do protocol testing. Network simulation techniques could be used to develop a simulation model of the communication system domain. Figure A-9 shows a domain model for a simulated MPS-based distributed application.

Multiple MPPMs would also have to be accessible to the simulation model of a heterogeneous distributed system. SDMPS sub-paths could contain function calls that accepted an MPPM parameter to obtain ETEs for the correct platform rather than have hard-coded MPPM ETEs in the SDMPS.

VI. CONCLUSIONS AND RECOMMENDATIONS

The investigation presented in this thesis was concerned with two questions related to avoiding the probe effect in testing real-time software implemented in high-level languages (HLL) developed using Object-Oriented Analysis (OOA). The uniprocessor question is whether it is feasible to develop a simulation model of an application-independent software-architecture and environment in which to execute the same HLL application-level binary code as would execute in a uniprocesor-based execution-environment. The distributed-systems question is whether it is feasible to extend the approach to distributed real-time applications.

The first section of this chapter summarizes the investigation and states the conclusions. The second section discusses the implications of this research for related subject areas and makes recommendations for further research.

A. SUMMARY AND CONCLUSIONS

Implementations of the same software architecture (SA) that generate the same program results with the same application-level binary-code have application-binary-code consistency (ABCC). This investigation employed a message-passing subsystem (MPS) software-architecture and a discrete-event simulation (DES) model of the MPS (SMPS) to investigate the uniprocessor question. An SMPS that has ABCC with the MPS supports application-level software testing conducted under controllable conditions free of the probe effect.

The study used the MPS to implement a set of state-machine execution-engines that execute OOA state-model actions (SMA). It restricted application-level SMAs to using the

MPS interface for all timing and synchronization operations. The approach also imposed a data-access policy and a synchronization-policy that assured mutually exclusive access to relevant data by each active object for the duration of its SMA.

The MPS design resolved the conflicting views of time between a real application and a DES-application. The MPS design restricted the SMA view of time to that provided by a logical clock that was only updated to "actual time" at the beginning of each SMA execution. SMA event-transmission was postponed until SMA completion in both the MPS and the SMPS. These two design properties supported ABCC by precluding the necessity to factor SMA application-code into multiple DES events for the SMPS.

The SMPS used an MPS platform-performance model (MPPM). The MPPM contained a model of the system clock function resolution for the target platform. All SMPS functions that accessed the system clock function relied on this model of clock resolution. The MPPM also contained a set of functions that returned MPS execution-time estimates (ETE) that model the execution-times of MPS module sub-paths under MPS-application execution conditions in a target environment. The study used a dual-loop (DL) approach to acquire the MPPM ETEs for the target execution-environment. The SMPS sub-paths contained hard-coded MPPM ETEs that were accumulated on-the-fly to obtain SMPS operation execution-time estimates. The SMPS used the ETEs to schedule the completion of SMPS SAs as DES events.

The investigation developed two MPS-based applications to investigate SMPS timing, synchronization, and the SMPS model of a preemptive multi-threaded run-time system. The applications had well-understood synchronization and timing properties. Each demonstration program ideal case was defined as hypothetical program execution on a platform with a perfect system clock and no execution overhead. The SMPS-based implementation of a uniprocessor model exhibited the desired behavior during the observation periods by only deviating from the ideal case by the amount expected for the MPPM.

The results of the investigation indicated that it is feasible to develop a DES implementation of a SA that has ABCC with an implementation designed for a uniprocessor execution-environment. However, several conditions must be met. First, program sub-path execution-times must be predictable in the target environment. Second, the program sub-path execution times of the real implementation must be available as a platform-performance-model to be incorporated into the simulation model. Finally, given a valid platform-performance-model, it must be possible to duplicate the timing properties of the real program in a simulation model.

The distributed-systems question did not add new research questions, so no separate investigation was conducted. There are two problems in the distributed-systems question. The first is how to model a multiprocessor-based single-platform system. The second is how to model multiple heterogeneous communicating platforms. The techniques developed in this investigation for the uniprocessor case could be extended to address the first problem. Network simulation techniques could be used to address the second problem.

The research reported in this thesis examined feature timing using an empirical approach. A need for realistic timing motivated this approach, but any empirical approach is limited by the conditions under which the study is conducted. This approach is expected to scale well to larger subsets of the Java language, and may apply readily to other languages, but this has not yet been proven. The results reported in this thesis offer a direction of inquiry, not a conclusive argument for applicability to any specific project.

B. RECOMMENDATIONS

The approach developed in this investigation allows the same HLL application-level binary-code to execute in both a target execution-environment and in a simulation model. This property enables software test designers to disregard the intrusive effects of source code instrumentation in application-level software testing. However, the benefit of this approach is not necessarily limited to testing application-level software. It extends to any service-domain that uses the modelled domain and can also be used for service-domain and application integration testing.

The approach will also add value to OOA-based software development environments that automatically generate source code from OOA models. The SMM *BridgePoint* toolset from Project Technology[1] uses an application-instance-to-architecture-instance mapping and a system-construction engine to generate applications from OOA models [Shlaer and Mellor 1997]. *BridgePoint* contains a model verifier that executes the state models in an OOA model before code generation. The approach developed in this investigation allows designers to test the correctness of their application-instance-to-architecture-instance mappings in an environment free of the probe effect by executing the generated code in a simulation model.

This approach is also useful in distributed systems and network-protocol software development. Network-protocol error-recovery software makes frequent use of timeouts to detect message-delivery failures. Higher performance protocols have tighter timing constraints and are often very time-sensitive. Network-protocol design often involves developing a separate simulation model using network simulation tools such as OPNET[2] or *ns*[3].

1. Project Technology, Inc. 7400 N. Oracle Rd. Suite 365 Tucson, AZ 85704, http://www.projtech.com

However, the approach presented in this thesis can be used to run the actual protocol and application code in the simulation model. This avoids questions about differences between the protocol software and the protocol simulation and only requires a single protocol implementation.

This investigation employed an SA-based approach that is essentially language-independent. However, language-specific SA-independent approaches are also worth considering [Huang et al. 1983]. This may be achieved in Java by developing a Java class-file compiler that replaces the *Thread*-related bytecodes in class-files with simulated Thread bytecodes. This approach would require developing a bytecode-level MPPM and instrumenting the class-files with CPU-time accumulation operations as part of class-file compilation. This approach would be widely applicable to Java programs. It would also require no source-code level modifications and has the advantage that only a single DL test-suite would need to be developed for the set of Java byte-codes. However, additional work would be needed to extend the approach to encompass many of the standard Java libraries.

2. MIL 3, Inc. · 3400 International Drive, NW · Washington, DC · 20008. http://www.mil3.com/products/modeler/simcycle.html.
3. UCB/LBNL/VINT Network Simulator - ns (version 2), http://mash.cs.berkeley.edu/ns/ns.html

LIST OF REFERENCES

Altman, N., "Factors Causing Unexpected Variation in Ada Benchmarks," CMU/SEI-87-TR-22, Software Engineering Instituted, Carnegie Mellon University, Pittsburgh, PA, October, 1987.

Altman, N. and Weiderman, N., "Timing Variations in Dual Loop Benchmarks," CMU/SEI-87-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, October, 1987.

Anderson, I., *A First Course in Combinatorial Mathematics*, Oxford University Press, Glasgow, Scotland, 1974.

Bell, R., "Code Generation from Object Models," *Embedded Systems Programming*, March, 1998.

Birstwistle, G., Dahl, O., Myrhoug, B., and Nygaard, K., *Simula Begin*, Studentliteratur (Lund) and Aurbach, New York, NY, 1973.

Burns, A., and Wellings, A., *Concurrency in Ada*, Cambridge University Press, Cambridge, UK, 1995.

Clapp, R., Duchensneau, R., Mudge, T., and Schultze, T., "Toward Real-Time Performance Benchmarks for Ada," *Communications of the ACM*, 29(8), pp. 760-778, August 1986.

Clark, L., Podgurski, A., Richardson, D., and Zeil, S., "A Formal Evaluation of Data Flow Path Selection Criteria," *IEEE Transactions on Software Engineering*, 15(11), pp. 1318-1332, November, 1989.

Dijkstra, E., "Hierarchical Ordering of Sequential Processes," *Acta Informatica*, 1, 1971.

Fidge, C., "Fundamentals of Distributed System Observation (version 1.1)", Technical Report No. 93-15. Software Verification Research Center, Department of Computer Science, The University of Queensland, Queensland, Australia, November 1993.

Gait, J., "A Probe Effect in Concurrent Programs," *Software–Practice and Experience,* 15, 6, 539-554.

Gosling, J., Joy, B., and Steele, G., *The Java Language Specification*, Addison-Wesley, Reading, MA, 1996.

Graybeal, W., and Pooch, U., *Simulation: Principles and Methods*, Winthrop Publishers Inc., Cambridge, MA, 1980.

Gupta, R., Jain,V., and Spezialetti, M., "An Approach for Monitoring Intrusion Removal in Real Time Systems," in *Proceedings of the 17th Real-Time Systems Symposium*, in Washington, DC, December 4-6, 1996.

Huang, J., Ho, M., and Law, T., "A Simulator for Real-time Software Debugging and Testing," *Software–Practice and Experience,* 14(9), 845-855, September 1984.

Jefferson, D. R., "Virtual Time," *ACM Transactions on Programming Languages and Systems*, 7(3):404-"425, July, 1985.

Jorgenson, P., *Software Testing: A Craftsman's Approach*, CRC Press, New York, NY, 1995.

Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, 21(7), pp. 558-565, 1978.

Lang, N., "Shlaer-Mellor Object-Oriented Analysis Rules," *Software Engineering Notes* ACM Press, 18(1), January 1993.

McDowell, C.E., and Helmbold, D.P., "Debugging Concurrent Programs," in *ACM Computing Surveys*, 21, 4, December 1989.

Meyers, G., *The Art of Software Testing*, John Wiley & Sons, New York, NY, 1979.

Netzer, R. and Miller, B., "Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs," Brown U. Technical Report CS-94-32, July 1994.

Ollerton, R., "Users Manual for the Ada Process-Oriented Simulation Library," *NOSC-TM-641*, 1992.

Pollack, R. and Campbell, D., "Clock Resolution and the PIWG Benchmark Suite," *Ada Letters Special Edition*, X(3), pp. 91-97, 1990.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

Russell, E., *Building Models with SIMSCRIPT II.5*, CACI, Los Angeles, CA, 1983.

Sha, L., and Goodenough, J., "Real-Time Scheduling Theory and Ada," *IEEE Computer*, April, 1990.

Shlaer, S., and Mellor, S., *Object-Oriented Systems Analysis–Modeling the World in Data*, Yourdon Press, Englewood Cliffs, NJ, 1988.

Shlaer, S., and Mellor, S., *Object LIfecycles–Modeling the World in States*, Yourdon Press, Englewood Cliffs, NJ, 1992.

Shlaer, S., and Mellor, S., "A Comparison of OOA and OMT," Project Technology, Berkeley, CA, August 1992.

Shlaer, S., and Mellor, S., "The Shlaer-Mellor Method," Project Technology, Berkeley, CA, 1993.

Shlaer, S., and Mellor, S., "Shlaer-Mellor Method: The OOA96 Report," Project Technology, Berkeley, CA, 1996.

Shlaer, S., and Mellor, S., "Synchronous Services," Project Technology, Berkeley, CA, August 1996.

Shlaer, S., and Mellor, S., "Bridges and Wormholes," Project Technology, Berkeley, CA, August 1996.

Shlaer, S., and Mellor, S., "Recursive Design of an Application-Independent Architecture," *IEEE Software*, January 1997.

Shutz, W., *The Testability of Distributed Real-Time Systems*, Kluwer Academinc Publishers, Boston, MA, 1993.

Tannenbaum, A. S., *Computer Networks*, Prentice Hall PTR, Upper Saddle River, N. J., 1996.

Tolmach, A.P. and Appel, A.W., "Debuggable Concurrency Extensions for Standard ML," CS-TR-352-91, Prinston University, October 1991.

Vestal,S., "Linear Benchmarks," *Ada Letters*, X(8), Novermber/December 1990.

Yellin, F. and Lindholm, T., *The Java Virtual Machine*, Addison-Wesley, Reading, MA. 1996.

APPENDIX A. MPS AND SMPS OOA MODEL DIAGRAMS

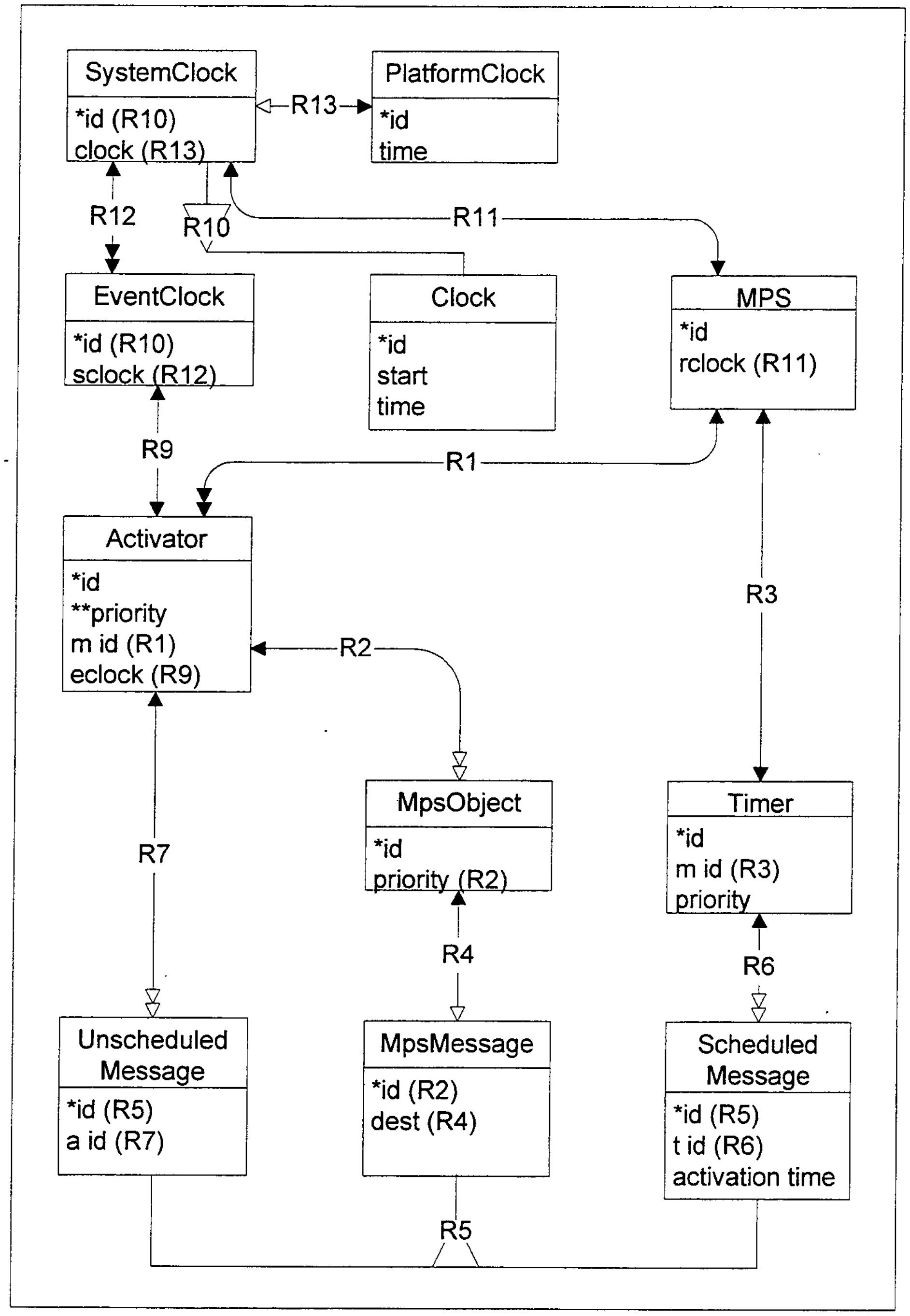

Figure A-1. MPS Object Information Model

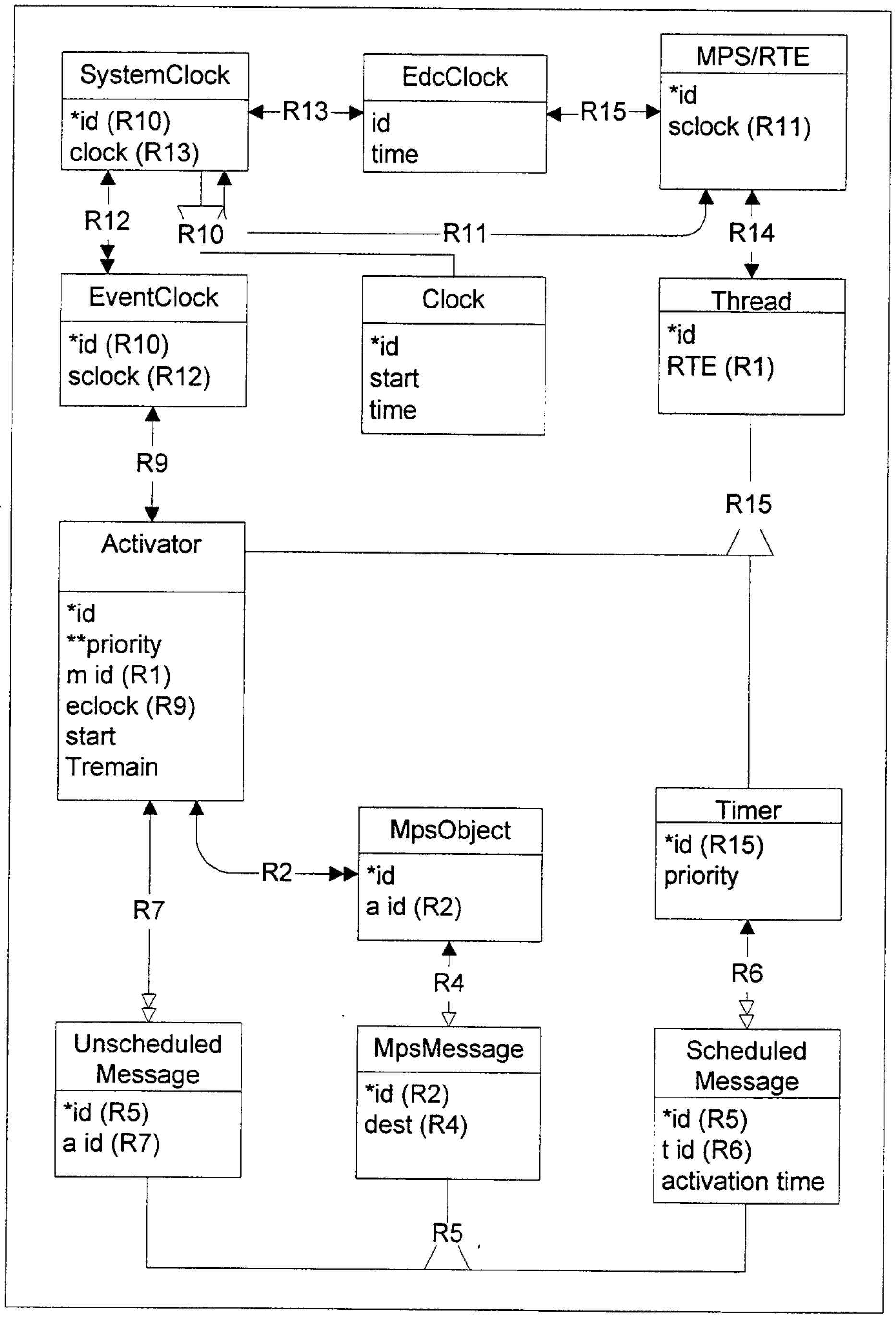

Figure A-2. SMPS Object Information Model

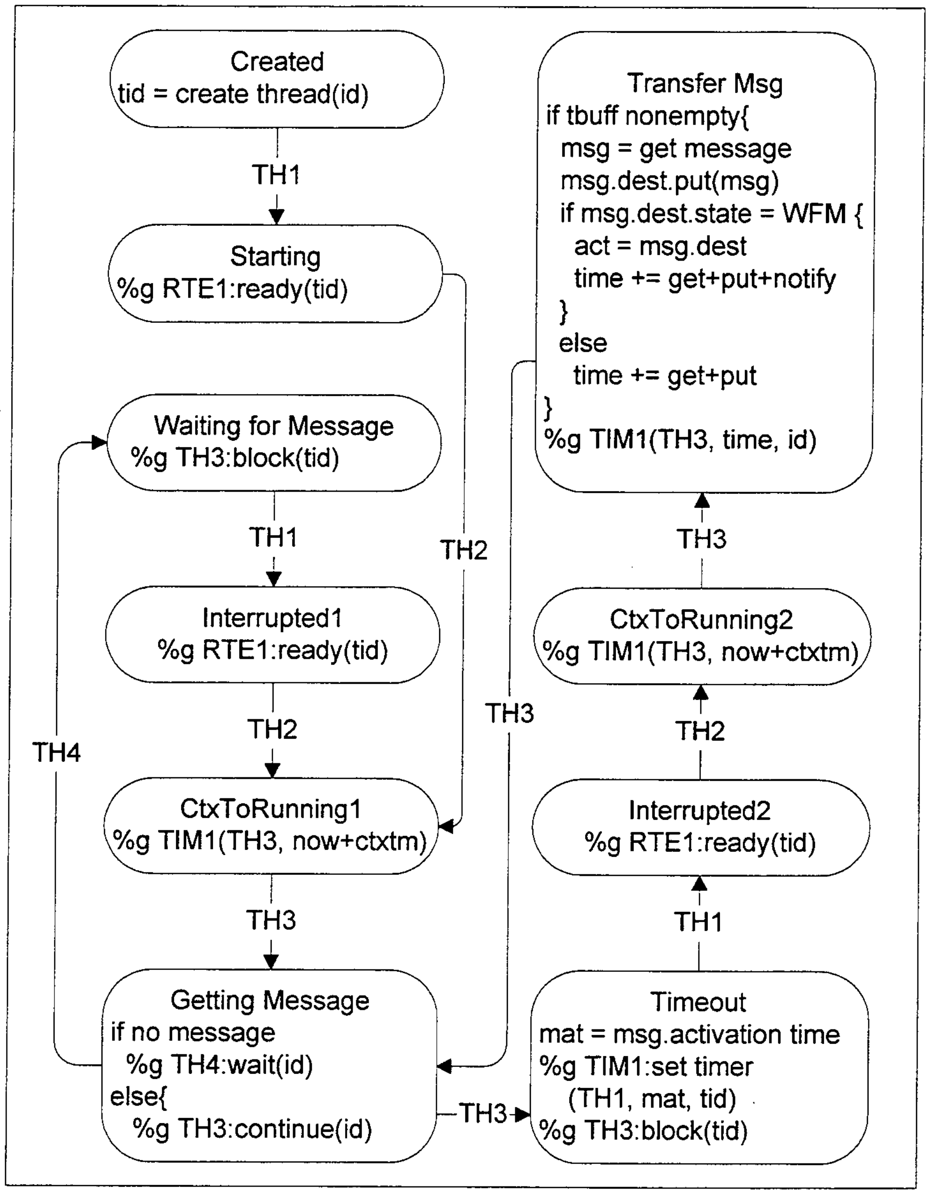

Figure A-3. SMPS Timer State Model

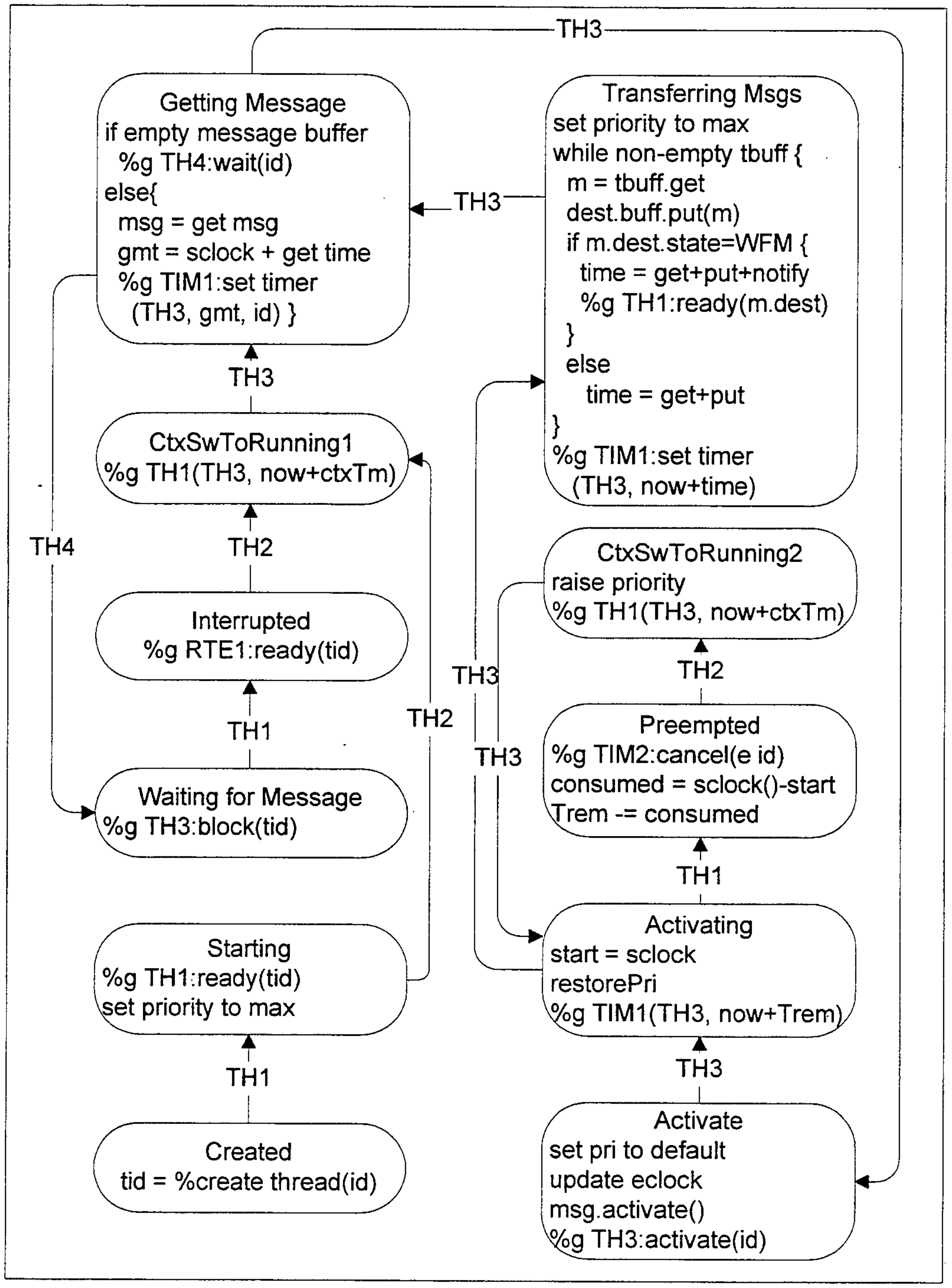

Figure A-4. SMPS Activator State Model

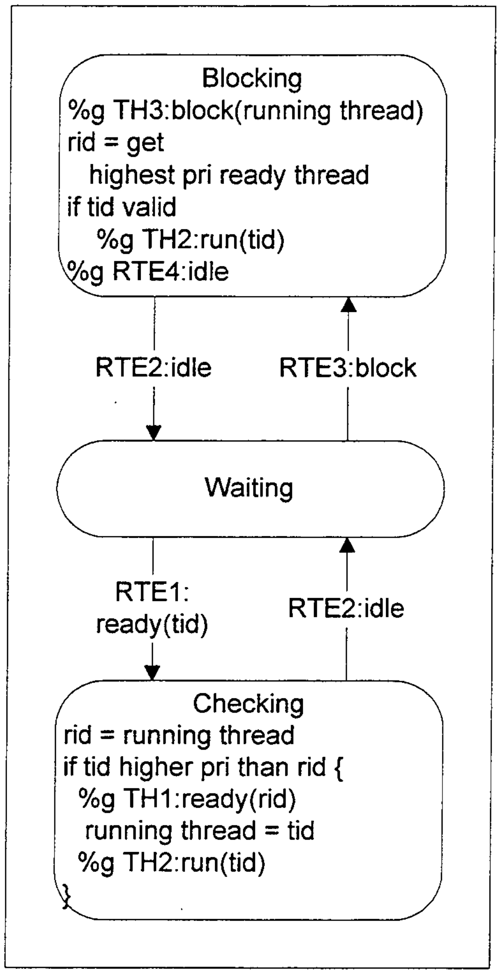

Figure A-5. SMPS Run-Time Executive State Model

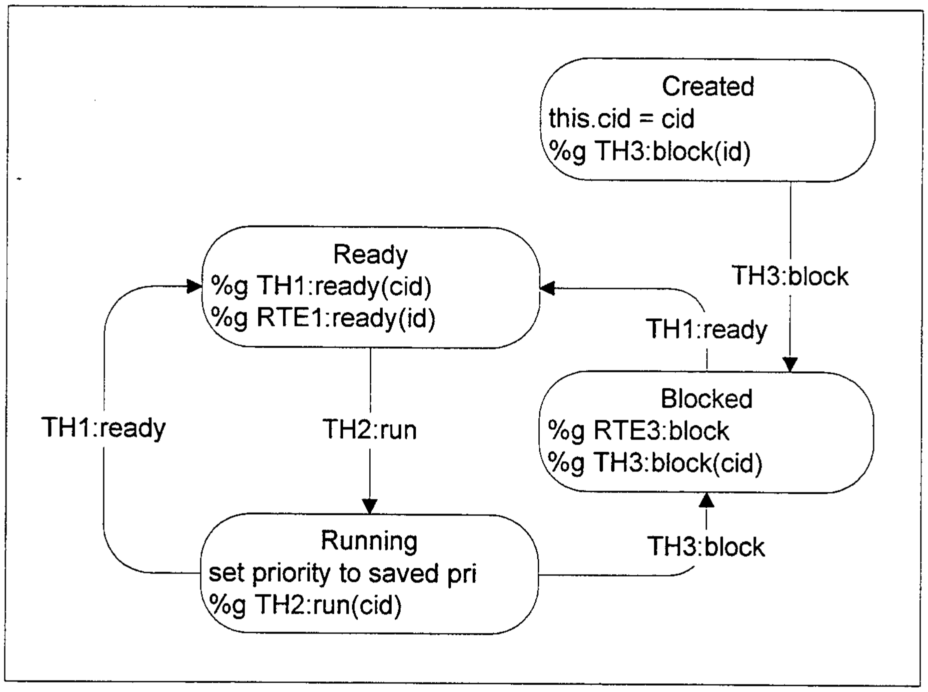

Figure A-6. SMPS Thread State Model

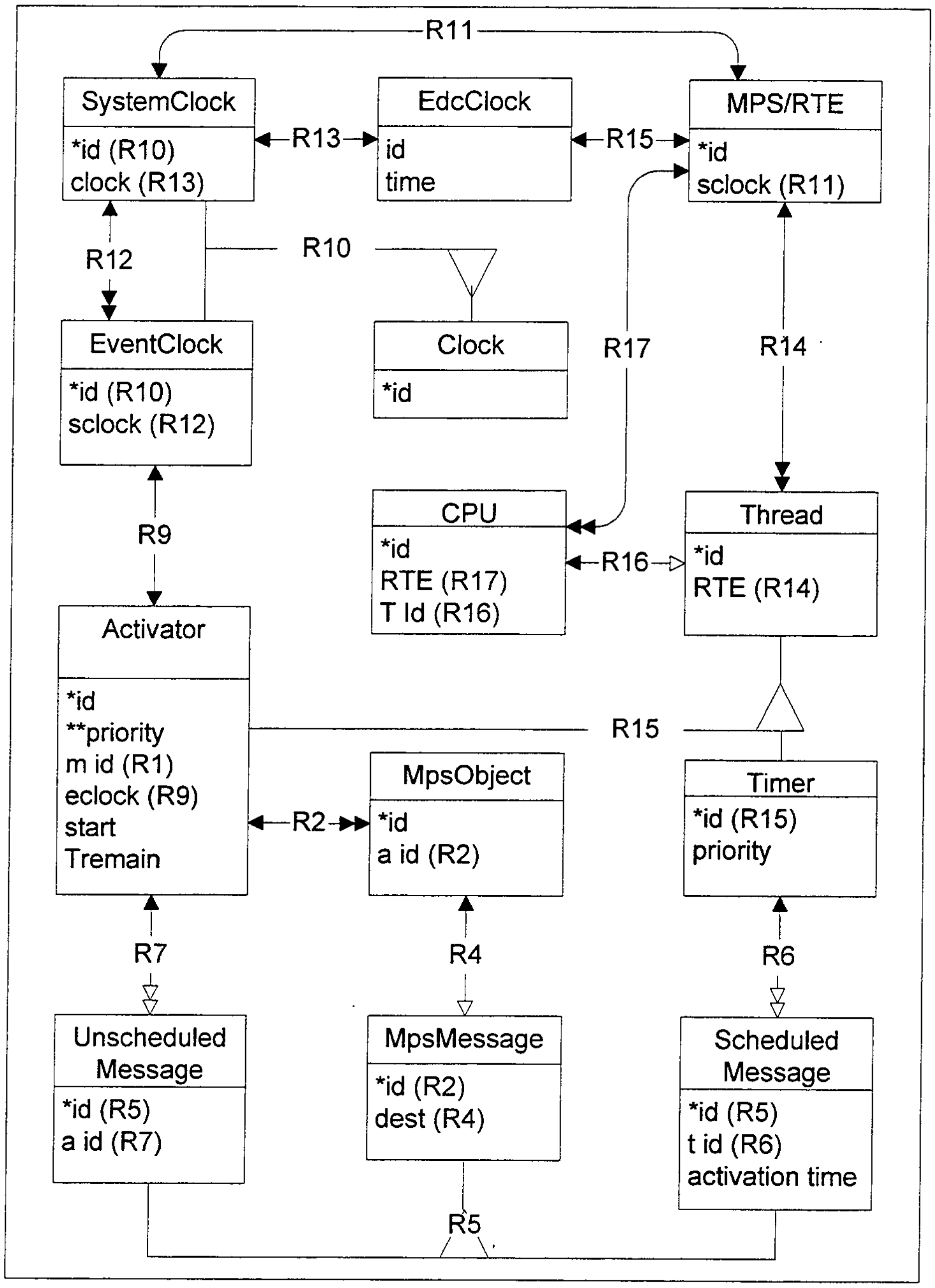

Figure A-7. Multiprocessor-based MPS Object Information Model

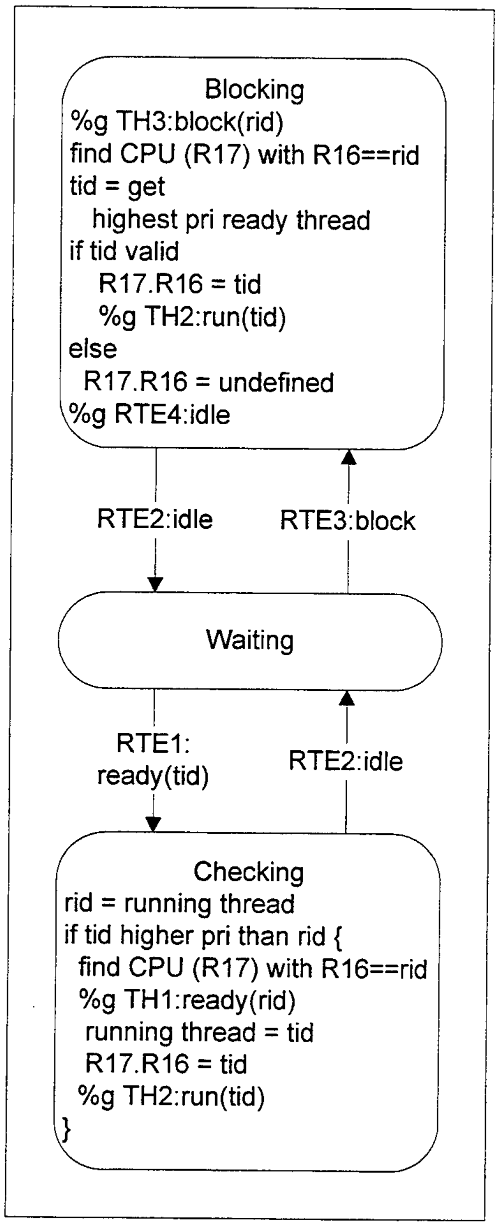

Figure A-8. Multiprocessor-based MPS Run-Time Executive State Model

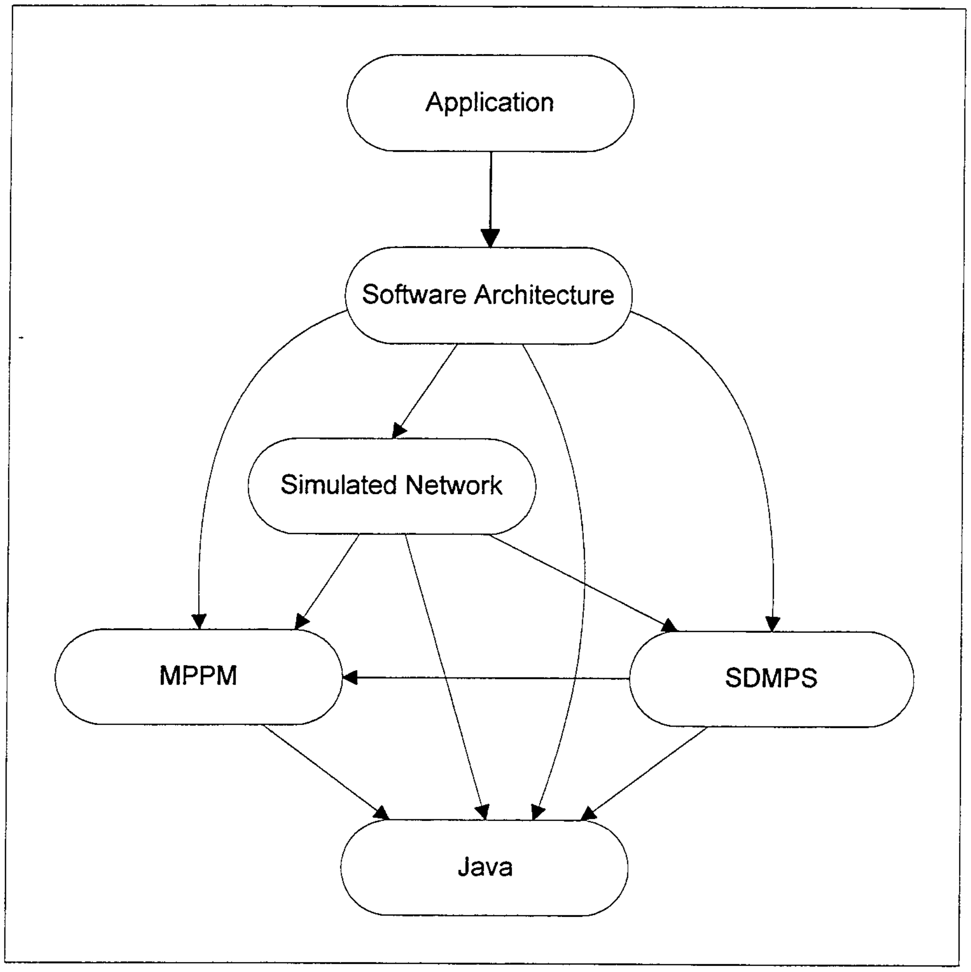

Figure A-9. Domain Model for a Simulated MPS-based Distributed Application

APPENDIX B. MPPM SOURCE CODE EXAMPLES

```
public final class ClockRes2
{
   static long[] time = new long[1000000];

   public static void main(String args[])
   {
      int count      = 0;
      double sum     = 0.0;
      for (int test=0; test<10; test++)
      {
         for (int idx=0; idx<time.length; idx++)
           time[idx] = System.currentTimeMillis();
         for (int idx=1; idx<time.length; idx++)
         {
            if (time[idx] > time[idx-1])
            {
               count++;
              long diff = time[idx] - time[idx-1];
               sum = sum + (double)(time[idx]
                                   - time[idx-1]);
            }
         }
      }
      System.out.println("resolution: "
                   + (sum/(double)count) + " ms");
   }
}
```

Figurre B-1. Program to Determine the Resolution of the Fundamental Time Unit Returned by Java *System.currentTimeMillis*

```
/** Get the current system time.
*
* @return current system time in milliseconds.
*/
public long millis()
{
   double time = Clock.get() //sim clock;
   time = Math.floor(Tcr*Math.floor(time/Tcr));
   return (long) time;
}
```

Figure B-2. The SMPS Model of the Java *System.currentTimeMillis* Method in class *MpsRealClock*

```
public final class DelayResolution extends Thread
{
    static final int N = 50, trials = 30;
    static final long []
    time = new long [N * trials],
    mtime = new long [N * trials];
    static int ctr = 0;
    public static void main(String args[])
    {
       object t = new object();
       for (int idx = 0; idx < N; idx++)
           for (int trial = 0; trial < trials; trial++)
            {
                rtime[ctr] = (long )(idx + 1);
                mtime[ctr] = t.Wait(rtime[ctr]);
                ctr++;
            }
       System.out.println("0, ");
       for (int idx = 0; idx < rtime.length; idx++)
           System.out.println(""
               + rtime[idx] + ", " + mtime[idx]);
    }
}
class object extends Object
{
    public synchronized long Wait(long time)
    {
       long start = 0;
       try
       {
           start = System.currentTimeMillis();
           wait(time);
           return System.currentTimeMillis() - start;
       }
       catch (InterruptedException e)
       {
           System.out.println(e.toString());
           return -1;
       }
    }
}
```

Figure B-3. Program to Examine the Resolution of the Java *wait* Statement

```
//%%/** Execution time estimate of 1st path.*/
//%%private static final double
//%%Vappend_1mf = "method.mipubv1pMf"
//%%           + "cond.iVbEqcVbMf"
//%%           + 2*"assign.iloloMf"
//%%           + "MpsNodeTO.updateNextNullMf";
//%%/** Execution time estimate of 2nd path.*/
//%%private static final double
//%%Vappend_2mf = "method.mipubv1pMf"
//%%           + "cond.iVbEqcVbMf"
//%%           + "assign.iloloMf"
//%%           + "MpsNodeTO.updateNextMf"
//%%           + "MpsNodeTO.updateNextNullMf";
/** Appends MpsMessage to buffer.
* @param msg the message to be appended. */
final void append(CpuCharger cc, MpsMessage msg)
{
   if (front == null)
   {
       back = msg;
       front = msg;
       msg.updateNext(null);
       //%%cc.charge(Vappend_1mf);
   }
   else
   {
       back.updateNext(msg);
       msg.updateNext(null);
       back = msg;
       //%%cc.charge(Vappend_2mf);
   }
}
```

Figure B-4. *MpsFifo.append* Method Before ETE Inlining

```
/** Execution time estimate of 1st path.*/
private static final double
Vappend_1mf = 3.320018450419108E-4
           + 2.449154853820801E-4
           + 2*4.805326461791992E-4
           + 4.1353702545166016E-4;
/** Execution time estimate of 2nd path.*/
private static final double
Vappend_2mf = 3.320018450419108E-4
           + 2.449154853820801E-4
           + 4.805326461791992E-4
           + 5.21540641784668E-4
           + 4.1353702545166016E-4;
/** Appends MpsMessage to buffer.
* @param msg the message to be appended. */
final void append(CpuCharger cc, MpsMessage msg)
{
   if (front == null)
   {
       back = msg;
       front = msg;
       msg.updateNext(null);
       cc.charge(Vappend_1mf);
   }
   else
   {
       back.updateNext(msg);
       msg.updateNext(null);
       back = msg;
       cc.charge(Vappend_2mf);
   }
}
```

Figure B-5. *MpsFifo.append* Method After ETE Inlining

APPENDIX C. DINING PHILOSOPHERS OOA MODEL DIAGRAMS

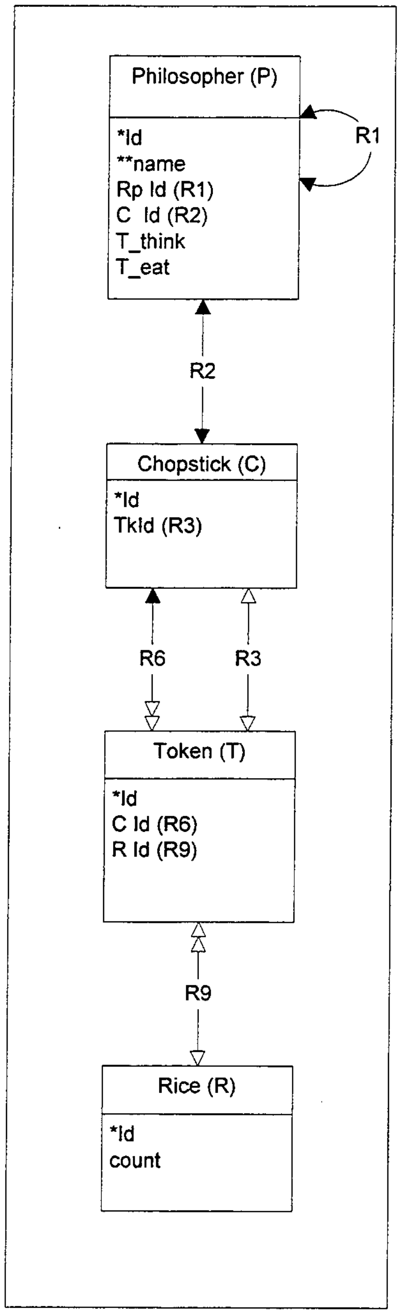

Figure C-1.
Dining Philosophers Object Information Model

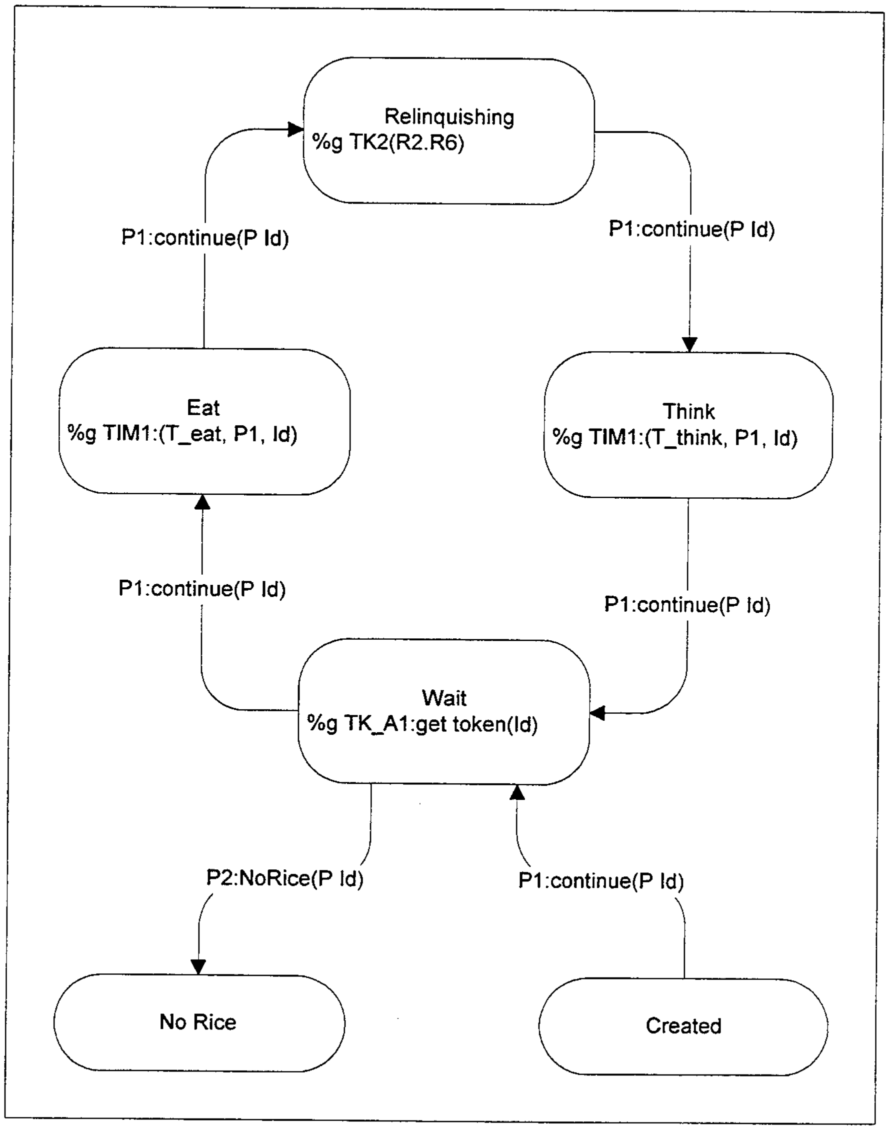

Figure C-2. Philosopher State Model

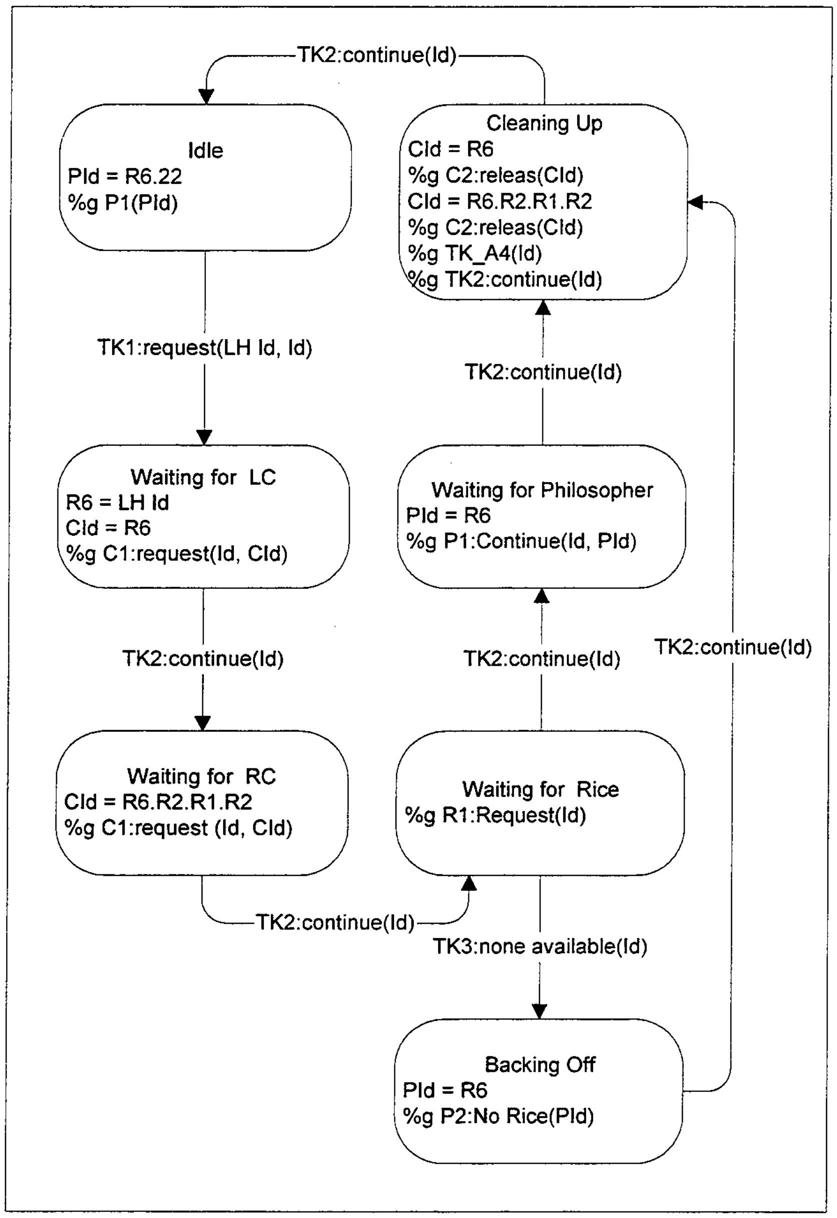

Figure C-3. Token State Model

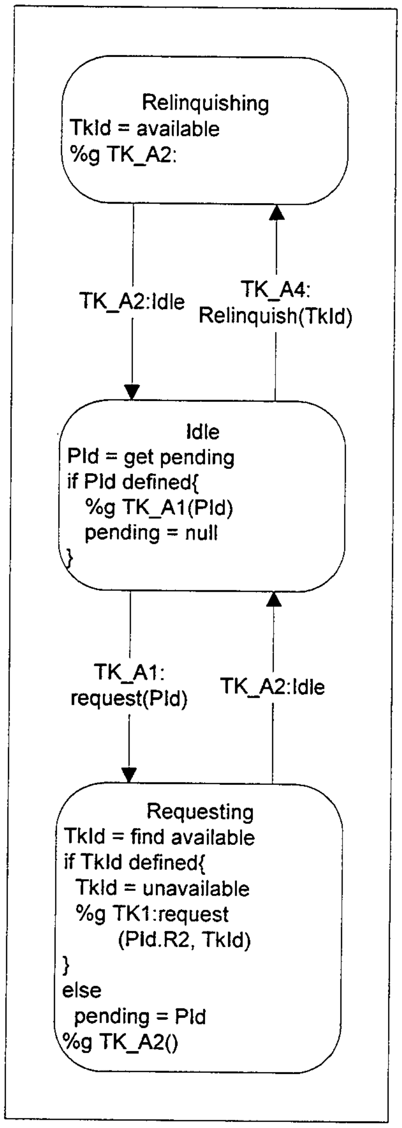

Figure C-4. Token Assigner State Model

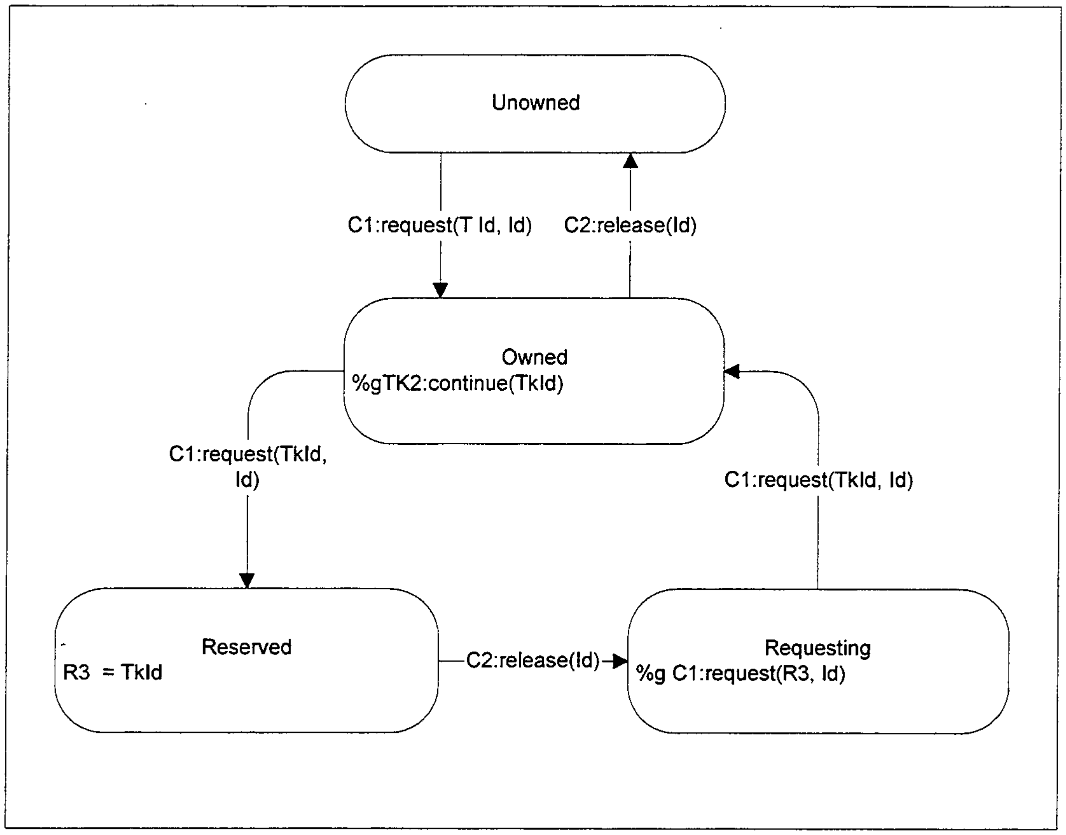

Figure C-5. Chopstick State Model

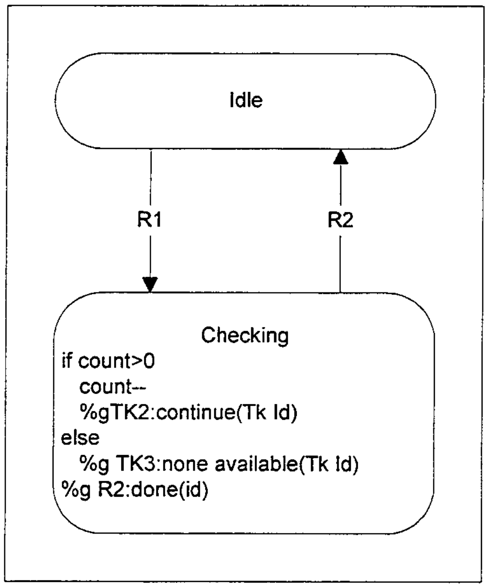

Figure C-6. Rice State Model

APPENDIX D. DINING PHILOSOPHER PROGRAM EXECUTION TRACES

D.1. APPLICATION-LEVEL TRACE

This section contains the complete execution trace for the application-level activities in the SMPS DP program. Each application-level trace statement has a line number, the simulation time in brackets, the object state, and may also have a message pertaining to the state.

```
1      [0.0000] TOK_A IDLE
2      [0.0000] Plato CREATED
3      [0.0000] Hegel CREATED
4      [0.0000] Decartes CREATED
5      [0.0000] Lao Tsu CREATED
6      [0.0000] Socrates CREATED
7      [1.4017] Plato WAITING: %g TK_A1(Plato) to Token_A
8      [1.4093] Hegel WAITING: %g TK_A1(Hegel) to Token_A
9      [1.4170] Decartes WAITING: %g TK_A1(Decartes) to Token_A
10     [1.4247] Lao Tsu WAITING: %g TK_A1(Lao Tsu) to Token_A
11     [1.4323] Socrates WAITING: %g TK_A1(Socrates) to Token_A
12     [1.4400] TOK_A REQUESTING: TOK_0 is avail: %g TK1(CS_0) to TOK_0
13     [1.4400] TOK_A IDLE
14     [1.4476] TOK_A REQUESTING: TOK_1 is avail: %g TK1(CS_1) to TOK_1
15     [1.4476] TOK_A IDLE
16     [1.4553] TOK_A REQUESTING: TOK_2 is avail: %g TK1(CS_2) to TOK_2
17     [1.4553] TOK_A IDLE
18     [1.4629] TOK_A REQUESTING: TOK_3 is avail: %g TK1(CS_3) to TOK_3
19     [1.4629] TOK_A IDLE
20     [1.4706] TOK_A REQUESTING: no avail tokens, Socrates pending
21     [1.4706] TOK_A IDLE
22     [1.4723] TOK_0 WAITING_FOR_LC: %g C1 to CS_0
23     [1.4799] TOK_1 WAITING_FOR_LC: %g C1 to CS_1
24     [1.4876] TOK_2 WAITING_FOR_LC: %g C1 to CS_2
25     [1.4952] TOK_3 WAITING_FOR_LC: %g C1 to CS_3
26     [1.5029] CS_0 OWNED: %g TK2 to TOK_0
27     [1.5105] CS_1 OWNED: %g TK2 to TOK_1
28     [1.5182] CS_2 OWNED: %g TK2 to TOK_2
29     [1.5258] CS_3 OWNED: %g TK2 to TOK_3
30     [1.5335] TOK_0 WAITING_FOR_RC: %g C1 to CS_4
31     [1.5411] TOK_1 WAITING_FOR_RC: %g C1 to CS_0
32     [1.5488] TOK_2 WAITING_FOR_RC: %g C1 to CS_1
33     [1.5564] TOK_3 WAITING_FOR_RC: %g C1 to CS_2
34     [1.5641] CS_4 OWNED: %g TK2 to TOK_0
35     [1.5718] CS_0 RESERVED
36     [1.5734] CS_1 RESERVED
37     [1.5751] CS_2 RESERVED
38     [1.5771] TOK_0 WAITING_FOR_RICE: %g R1 to Rice
39     [1.5851] RICE %g TK2 to TOK_0, 9 bites remaining
40     [1.5930] TOK_0 WAITING_FOR_PHILOSOPHER: %g P1 to Plato
41     [1.6010] Plato EAT: scheduling P1 to self at 150ms
42     [151.4095] Plato RELINQUISHING: %g TK2 to TOK_0
43     [151.4174] TOK_0 CLEANING_UP: %g C2 to CS_0
44     [151.4174] TOK_0 CLEANING_UP: %g C2 to CS_4
45     [151.4174] TOK_0 CLEANING_UP: %g TK_A4 to Token_A
```

```
46    [151.4421] CS_0 REQUESTING: %g C1 self-event
47    [151.4421] CS_0 OWNED: %g TK2 to TOK_1
48    [151.4497] CS_4 UNOWNED
49    [151.4514] TOK_A RELINQUISHING: TOK_0 avail, %g TK_A2 self-event
50    [151.4514] TOK_A IDLE: %g TK_A1(Socrates) to self
51    [151.4591] Plato THINK: scheduling P1 to self at 390ms
52    [151.4689] TOK_1 WAITING_FOR_RICE: %g R1 to Rice
53    [152.8766] TOK_A REQUESTING: TOK_0 is avail: %g TK1(CS_4) to TOK_0
54    [152.8766] TOK_A IDLE
55    [152.8842] RICE %g TK2 to TOK_1, 8 bites remaining
56    [152.8919] TOK_0 WAITING_FOR_LC: %g C1 to CS_4
57    [152.8995] TOK_1 WAITING_FOR_PHILOSOPHER: %g P1 to Hegel
58    [152.9072] CS_4 OWNED: %g TK2 to TOK_0
59    [152.9148] Hegel EAT: scheduling P1 to self at 290ms
60    [152.9251] TOK_0 WAITING_FOR_RC: %g C1 to CS_3
61    [154.3331] CS_3 RESERVED
62    [291.4098] Hegel RELINQUISHING: %g TK2 to TOK_1
63    [291.4177] TOK_1 CLEANING_UP: %g C2 to CS_1
64    [291.4177] TOK_1 CLEANING_UP: %g C2 to CS_0
65    [291.4177] TOK_1 CLEANING_UP: %g TK_A4 to Token_A
66    [291.4424] CS_1 REQUESTING: %g C1 self-event
67    [291.4424] CS_1 OWNED: %g TK2 to TOK_2
68    [291.4500] CS_0 UNOWNED
69    [291.4517] TOK_A RELINQUISHING: TOK_1 avail, %g TK_A2 self-event
70    [291.4517] TOK_A IDLE
71    [291.4534] Hegel THINK: scheduling P1 to self at 531ms
72    [291.4648] TOK_2 WAITING_FOR_RICE: %g R1 to Rice
73    [291.4728] RICE %g TK2 to TOK_2, 7 bites remaining
74    [291.4807] TOK_2 WAITING_FOR_PHILOSOPHER: %g P1 to Decartes
75    [291.4887] Decartes EAT: scheduling P1 to self at 431ms
76    [391.4098] Plato WAITING: %g TK_A1(Plato) to Token_A
77    [391.4177] TOK_A REQUESTING: TOK_1 is avail: %g TK1(CS_0) to TOK_1
78    [391.4177] TOK_A IDLE
79    [391.4257] TOK_1 WAITING_FOR_LC: %g C1 to CS_0
80    [391.4336] CS_0 OWNED: %g TK2 to TOK_1
81    [391.4415] TOK_1 WAITING_FOR_RC: %g C1 to CS_4
82    [391.4495] CS_4 RESERVED
83    [432.4098] Decartes RELINQUISHING: %g TK2 to TOK_2
84    [432.4177] TOK_2 CLEANING_UP: %g C2 to CS_2
85    [432.4177] TOK_2 CLEANING_UP: %g C2 to CS_1
86    [432.4177] TOK_2 CLEANING_UP: %g TK_A4 to Token_A
87    [432.4424] CS_2 REQUESTING: %g C1 self-event
88    [432.4424] CS_2 OWNED: %g TK2 to TOK_3
89    [432.4500] CS_1 UNOWNED
90    [432.4517] TOK_A RELINQUISHING: TOK_2 avail, %g TK_A2 self-event
91    [432.4517] TOK_A IDLE
92    [432.4534] Decartes THINK: scheduling P1 to self at 671ms
93    [432.4648] TOK_3 WAITING_FOR_RICE: %g R1 to Rice
94    [432.4728] RICE %g TK2 to TOK_3, 6 bites remaining
95    [432.4807] TOK_3 WAITING_FOR_PHILOSOPHER: %g P1 to Lao Tsu
96    [432.4887] Lao Tsu EAT: scheduling P1 to self at 671ms
97    [532.4098] Hegel WAITING: %g TK_A1(Hegel) to Token_A
98    [532.4177] TOK_A REQUESTING: TOK_2 is avail: %g TK1(CS_1) to TOK_2
99    [532.4177] TOK_A IDLE
100   [532.4257] TOK_2 WAITING_FOR_LC: %g C1 to CS_1
101   [532.4336] CS_1 OWNED: %g TK2 to TOK_2
102   [532.4415] TOK_2 WAITING_FOR_RC: %g C1 to CS_0
103   [532.4495] CS_0 RESERVED
104   [672.4098] Lao Tsu RELINQUISHING: %g TK2 to TOK_3
105   [673.8227] Decartes WAITING: %g TK_A1(Decartes) to Token_A
106   [673.8304] TOK_3 CLEANING_UP: %g C2 to CS_3
107   [673.8304] TOK_3 CLEANING_UP: %g C2 to CS_2
108   [673.8304] TOK_3 CLEANING_UP: %g TK_A4 to Token_A
109   [673.8551] TOK_A REQUESTING: no avail tokens, Decartes pending
```

```
110   [673.8551] TOK_A IDLE
111   [673.8568] CS_3 REQUESTING: %g C1 self-event
112   [673.8568] CS_3 OWNED: %g TK2 to TOK_0
113   [673.8644] CS_2 UNOWNED
114   [673.8661] TOK_A RELINQUISHING: TOK_3 avail, %g TK_A2 self-event
115   [673.8661] TOK_A IDLE: %g TK_A1(Decartes) to self
116   [673.8737] Lao Tsu THINK: scheduling P1 to self at 1171ms
117   [673.8836] TOK_0 WAITING_FOR_RICE: %g R1 to Rice
118   [675.2912] TOK_A REQUESTING: TOK_3 is avail: %g TK1(CS_2) to TOK_3
119   [675.2912] TOK_A IDLE
120   [675.2989] RICE %g TK2 to TOK_0, 5 bites remaining
121   [675.3066] TOK_3 WAITING_FOR_LC: %g C1 to CS_2
122   [675.3142] TOK_0 WAITING_FOR_PHILOSOPHER: %g P1 to Socrates
123   [675.3219] CS_2 OWNED: %g TK2 to TOK_3
124   [675.3295] Socrates EAT: scheduling P1 to self at 921ms
125   [675.3398] TOK_3 WAITING_FOR_RC: %g C1 to CS_1
126   [676.7478] CS_1 RESERVED
127   [922.4098] Socrates RELINQUISHING: %g TK2 to TOK_0
128   [922.4177] TOK_0 CLEANING_UP: %g C2 to CS_4
129   [922.4177] TOK_0 CLEANING_UP: %g C2 to CS_3
130   [922.4177] TOK_0 CLEANING_UP: %g TK_A4 to Token_A
131   [922.4424] CS_4 REQUESTING: %g C1 self-event
132   [922.4424] CS_4 OWNED: %g TK2 to TOK_1
133   [922.4500] CS_3 UNOWNED
134   [922.4517] TOK_A RELINQUISHING: TOK_0 avail, %g TK_A2 self-event
135   [922.4517] TOK_A IDLE
136   [922.4534] Socrates THINK: scheduling P1 to self at 1421ms
137   [922.4648] TOK_1 WAITING_FOR_RICE: %g R1 to Rice
138   [922.4728] RICE %g TK2 to TOK_1, 4 bites remaining
139   [922.4807] TOK_1 WAITING_FOR_PHILOSOPHER: %g P1 to Plato
140   [922.4887] Plato EAT: scheduling P1 to self at 1071ms
141   [1072.4098] Plato RELINQUISHING: %g TK2 to TOK_1
142   [1072.4177] TOK_1 CLEANING_UP: %g C2 to CS_0
143   [1072.4177] TOK_1 CLEANING_UP: %g C2 to CS_4
144   [1072.4177] TOK_1 CLEANING_UP: %g TK_A4 to Token_A
145   [1072.4424] CS_0 REQUESTING: %g C1 self-event
146   [1072.4424] CS_0 OWNED: %g TK2 to TOK_2
147   [1072.4500] CS_4 UNOWNED
148   [1072.4517] TOK_A RELINQUISHING: TOK_1 avail, %g TK_A2 self-event
149   [1072.4517] TOK_A IDLE
150   [1072.4534] Plato THINK: scheduling P1 to self at 1312ms
151   [1072.4654] TOK_2 WAITING_FOR_RICE: %g R1 to Rice
152   [1072.4733] RICE %g TK2 to TOK_2, 3 bites remaining
153   [1072.4813] TOK_2 WAITING_FOR_PHILOSOPHER: %g P1 to Hegel
154   [1072.4892] Hegel EAT: scheduling P1 to self at 1212ms
155   [1172.4098] Lao Tsu WAITING: %g TK_A1(Lao Tsu) to Token_A
156   [1172.4177] TOK_A REQUESTING: TOK_0 is avail: %g TK1(CS_3) to TOK_0
157   [1172.4177] TOK_A IDLE
158   [1172.4257] TOK_0 WAITING_FOR_LC: %g C1 to CS_3
159   [1172.4336] CS_3 OWNED: %g TK2 to TOK_0
160   [1172.4415] TOK_0 WAITING_FOR_RC: %g C1 to CS_2
161   [1172.4495] CS_2 RESERVED
162   [1213.4098] Hegel RELINQUISHING: %g TK2 to TOK_2
163   [1213.4177] TOK_2 CLEANING_UP: %g C2 to CS_1
164   [1213.4177] TOK_2 CLEANING_UP: %g C2 to CS_0
165   [1213.4177] TOK_2 CLEANING_UP: %g TK_A4 to Token_A
166   [1213.4424] CS_1 REQUESTING: %g C1 self-event
167   [1213.4424] CS_1 OWNED: %g TK2 to TOK_3
168   [1213.4500] CS_0 UNOWNED
169   [1213.4517] TOK_A RELINQUISHING: TOK_2 avail, %g TK_A2 self-event
170   [1213.4517] TOK_A IDLE
171   [1213.4534] Hegel THINK: scheduling P1 to self at 1453ms
172   [1213.4655] TOK_3 WAITING_FOR_RICE: %g R1 to Rice
173   [1213.4734] RICE %g TK2 to TOK_3, 2 bites remaining
```

```
174    [1213.4814] TOK_3 WAITING_FOR_PHILOSOPHER: %g P1 to Decartes
175    [1213.4893] Decartes EAT: scheduling P1 to self at 1353ms
176    [1313.4098] Plato WAITING: %g TK_A1(Plato) to Token_A
177    [1313.4177] TOK_A REQUESTING: TOK_1 is avail: %g TK1(CS_0) to TOK_1
178    [1313.4177] TOK_A IDLE
179    [1313.4257] TOK_1 WAITING_FOR_LC: %g C1 to CS_0
180    [1313.4336] CS_0 OWNED: %g TK2 to TOK_1
181    [1313.4415] TOK_1 WAITING_FOR_RC: %g C1 to CS_4
182    [1313.4495] CS_4 OWNED: %g TK2 to TOK_1
183    [1313.4574] TOK_1 WAITING_FOR_RICE: %g R1 to Rice
184    [1313.4654] RICE %g TK2 to TOK_1, 1 bites remaining
185    [1313.4733] TOK_1 WAITING_FOR_PHILOSOPHER: %g P1 to Plato
186    [1313.4812] Plato EAT: scheduling P1 to self at 1462ms
187    [1354.4098] Decartes RELINQUISHING: %g TK2 to TOK_3
188    [1354.4177] TOK_3 CLEANING_UP: %g C2 to CS_2
189    [1354.4177] TOK_3 CLEANING_UP: %g C2 to CS_1
190    [1354.4177] TOK_3 CLEANING_UP: %g TK_A4 to Token_A
191    [1354.4424] CS_2 REQUESTING: %g C1 self-event
192    [1354.4424] CS_2 OWNED: %g TK2 to TOK_0
193    [1354.4500] CS_1 UNOWNED
194    [1354.4517] TOK_A RELINQUISHING: TOK_3 avail, %g TK_A2 self-event
195    [1354.4517] TOK_A IDLE
196    [1354.4534] Decartes THINK: scheduling P1 to self at 1593ms
197    [1354.4655] TOK_0 WAITING_FOR_RICE: %g R1 to Rice
198    [1354.4734] RICE %g TK2 to TOK_0, 0 bites remaining
199    [1354.4814] TOK_0 WAITING_FOR_PHILOSOPHER: %g P1 to Lao Tsu
200    [1354.4893] Lao Tsu EAT: scheduling P1 to self at 1593ms
201    [1422.4098] Socrates WAITING: %g TK_A1(Socrates) to Token_A
202    [1422.4177] TOK_A REQUESTING: TOK_2 is avail: %g TK1(CS_4) to TOK_2
203    [1422.4177] TOK_A IDLE
204    [1422.4257] TOK_2 WAITING_FOR_LC: %g C1 to CS_4
205    [1422.4336] CS_4 RESERVED
206    [1454.4098] Hegel WAITING: %g TK_A1(Hegel) to Token_A
207    [1454.4177] TOK_A REQUESTING: TOK_3 is avail: %g TK1(CS_1) to TOK_3
208    [1454.4177] TOK_A IDLE
209    [1454.4257] TOK_3 WAITING_FOR_LC: %g C1 to CS_1
210    [1454.4336] CS_1 OWNED: %g TK2 to TOK_3
211    [1454.4415] TOK_3 WAITING_FOR_RC: %g C1 to CS_0
212    [1454.4495] CS_0 RESERVED
213    [1463.4098] Plato RELINQUISHING: %g TK2 to TOK_1
214    [1463.4177] TOK_1 CLEANING_UP: %g C2 to CS_0
215    [1463.4177] TOK_1 CLEANING_UP: %g C2 to CS_4
216    [1463.4177] TOK_1 CLEANING_UP: %g TK_A4 to Token_A
217    [1463.4424] CS_0 REQUESTING: %g C1 self-event
218    [1463.4424] CS_0 OWNED: %g TK2 to TOK_3
219    [1463.4500] CS_4 REQUESTING: %g C1 self-event
220    [1463.4500] CS_4 OWNED: %g TK2 to TOK_2
221    [1463.4577] TOK_A RELINQUISHING: TOK_1 avail, %g TK_A2 self-event
222    [1463.4577] TOK_A IDLE
223    [1463.4594] Plato THINK: scheduling P1 to self at 1703ms
224    [1463.4711] TOK_3 WAITING_FOR_RICE: %g R1 to Rice
225    [1463.4788] TOK_2 WAITING_FOR_RC: %g C1 to CS_3
226    [1463.4941] CS_3 RESERVED
227    [1463.4961] TOK_3 BACKING_OFF: %g P2 to Hegel
228    [1463.4961] TOK_3 CLEANING_UP: %g C2 to CS_1
229    [1463.4961] TOK_3 CLEANING_UP: %g C2 to CS_0
230    [1463.4961] TOK_3 CLEANING_UP: %g TK_A4 to Token_A
231    [1463.5264] Hegel NO_RICE
232    [1463.5281] CS_1 UNOWNED
233    [1463.5298] CS_0 UNOWNED
234    [1463.5315] TOK_A RELINQUISHING: TOK_3 avail, %g TK_A2 self-event
235    [1463.5315] TOK_A IDLE
236    [1594.4098] Lao Tsu RELINQUISHING: %g TK2 to TOK_0
237    [1595.8230] Decartes WAITING: %g TK_A1(Decartes) to Token_A
```

```
238    [1595.8307] TOK_0 CLEANING_UP: %g C2 to CS_3
239    [1595.8307] TOK_0 CLEANING_UP: %g C2 to CS_2
240    [1595.8307] TOK_0 CLEANING_UP: %g TK_A4 to Token_A
241    [1595.8554] TOK_A REQUESTING: TOK_1 is avail: %g TK1(CS_2) to TOK_1
242    [1595.8554] TOK_A IDLE
243    [1595.8630] CS_3 REQUESTING: %g C1 self-event
244    [1595.8630] CS_3 OWNED: %g TK2 to TOK_2
245    [1595.8707] CS_2 UNOWNED
246    [1595.8724] TOK_A RELINQUISHING: TOK_0 avail, %g TK_A2 self-event
247    [1595.8724] TOK_A IDLE
248    [1595.8741] Lao Tsu THINK: scheduling P1 to self at 2093ms
249    [1595.8852] TOK_1 WAITING_FOR_LC: %g C1 to CS_2
250    [1595.8928] TOK_2 WAITING_FOR_RICE: %g R1 to Rice
251    [1595.9005] CS_2 OWNED: %g TK2 to TOK_1
252    [1595.9158] TOK_1 WAITING_FOR_RC: %g C1 to CS_1
253    [1595.9234] TOK_2 BACKING_OFF: %g P2 to Socrates
254    [1595.9234] TOK_2 CLEANING_UP: %g C2 to CS_4
255    [1595.9234] TOK_2 CLEANING_UP: %g C2 to CS_3
256    [1595.9234] TOK_2 CLEANING_UP: %g TK_A4 to Token_A
257    [1595.9538] CS_1 OWNED: %g TK2 to TOK_1
258    [1595.9615] Socrates NO_RICE
259    [1595.9631] CS_4 UNOWNED
260    [1595.9648] CS_3 UNOWNED
261    [1595.9665] TOK_A RELINQUISHING: TOK_2 avail, %g TK_A2 self-event
262    [1595.9665] TOK_A IDLE
263    [1595.9702] TOK_1 WAITING_FOR_RICE: %g R1 to Rice
264    [1595.9861] TOK_1 BACKING_OFF: %g P2 to Decartes
265    [1595.9861] TOK_1 CLEANING_UP: %g C2 to CS_2
266    [1595.9861] TOK_1 CLEANING_UP: %g C2 to CS_1
267    [1595.9861] TOK_1 CLEANING_UP: %g TK_A4 to Token_A
268    [1596.0164] Decartes NO_RICE
269    [1596.0181] CS_2 UNOWNED
270    [1596.0198] CS_1 UNOWNED
271    [1596.0215] TOK_A RELINQUISHING: TOK_1 avail, %g TK_A2 self-event
272    [1596.0215] TOK_A IDLE
273    [1704.4098] Plato WAITING: %g TK_A1(Plato) to Token_A
274    [1704.4177] TOK_A REQUESTING: TOK_0 is avail: %g TK1(CS_0) to TOK_0
275    [1704.4177] TOK_A IDLE
276    [1704.4257] TOK_0 WAITING_FOR_LC: %g C1 to CS_0
277    [1704.4336] CS_0 OWNED: %g TK2 to TOK_0
278    [1704.4415] TOK_0 WAITING_FOR_RC: %g C1 to CS_4
279    [1704.4495] CS_4 OWNED: %g TK2 to TOK_0
280    [1704.4574] TOK_0 WAITING_FOR_RICE: %g R1 to Rice
281    [1704.4733] TOK_0 BACKING_OFF: %g P2 to Plato
282    [1704.4733] TOK_0 CLEANING_UP: %g C2 to CS_0
283    [1704.4733] TOK_0 CLEANING_UP: %g C2 to CS_4
284    [1704.4733] TOK_0 CLEANING_UP: %g TK_A4 to Token_A
285    [1704.5036] Plato NO_RICE
286    [1704.5053] CS_0 UNOWNED
287    [1704.5070] CS_4 UNOWNED
288    [1704.5087] TOK_A RELINQUISHING: TOK_0 avail, %g TK_A2 self-event
289    [1704.5087] TOK_A IDLE
290    [2094.4095] Lao Tsu WAITING: %g TK_A1(Lao Tsu) to Token_A
291    [2094.4174] TOK_A REQUESTING: TOK_0 is avail: %g TK1(CS_3) to TOK_0
292    [2094.4174] TOK_A IDLE
293    [2094.4254] TOK_0 WAITING_FOR_LC: %g C1 to CS_3
294    [2094.4333] CS_3 OWNED: %g TK2 to TOK_0
295    [2094.4412] TOK_0 WAITING_FOR_RC: %g C1 to CS_2
296    [2094.4492] CS_2 OWNED: %g TK2 to TOK_0
297    [2094.4571] TOK_0 WAITING_FOR_RICE: %g R1 to Rice
298    [2094.4730] TOK_0 BACKING_OFF: %g P2 to Lao Tsu
299    [2094.4730] TOK_0 CLEANING_UP: %g C2 to CS_3
300    [2094.4730] TOK_0 CLEANING_UP: %g C2 to CS_2
301    [2094.4730] TOK_0 CLEANING_UP: %g TK_A4 to Token_A
```

```
302    [2094.5033] Lao Tsu NO_RICE
303    [2094.5050] CS_3 UNOWNED
304    [2094.5067] CS_2 UNOWNED
305    [2094.5084] TOK_A RELINQUISHING: TOK_0 avail, %g TK_A2 self-event
306    [2094.5084] TOK_A IDLE
307    [2094.5104] Plato spent 450 ms eating 3 bites of rice and 750 ms
thinking
308    [2094.5104] Hegel spent 300 ms eating 2 bites of rice and 500 ms
thinking
309    [2094.5104] Decartes spent 300 ms eating 2 bites of rice and 500 ms
thinking
310    [2094.5104] Lao Tsu spent 500 ms eating 2 bites of rice and 1000 ms
thinking
311    [2094.5104] Socrates spent 250 ms eating 1 bites of rice and 500 ms
thinking
```

D.2 SMPS-LEVEL TRACES: CREATED TO WAIT

This section has the SMPS-level traces that underly execution indicated by lines six and seven shown in section D.1. SMPS-level trace statement have a line number, the simulation time in brackets and a message. The message does not always print object state.

```
1      [0000.0000] Socrates CREATED
2      [0000.0000] mps initSend: P1(Plato)
3      [0000.0000] mps initSend: P1(Hegel)
4      [0000.0000] mps initSend: P1(Decartes)
5      [0000.0000] mps initSend: P1(Lao Tsu)
6      [0000.0000] mps initSend: P1(Socrates)
7      [0000.0000] mps starting activators
8      [0000.0000] MpsTimer sending cp.mps.MpsTimer$TH1 to TIMER t:0.0
9      [0000.0000] mps starting activator 0
10     [0000.0000] MpsActivator0 TH1 to MpsActivator0 S[CREATED,BLOCKED]
p:9 t:0.0 delivered
11     [0000.0000] MpsActivator0 EXECUTING STARTING-action [READY] P:9
12     [0000.0000] MpsTimer executing STARTING [READY] pri:10
13     [0000.0000] MpsTimer sending TH2 to MpsTimer S[STARTING,READY] t:-
1.0
14     [0000.0000] MpsTimer executing CTXSW_TO_RUNNING1 [RUNNING] pri:10
15     [0000.7000] MpsTimer executing GETTING_MSG [RUNNING] pri:10
16     [0000.7000] MpsTimer executing WAITING_FOR_MSG [BLOCKED] pri:10
17     [0000.7000] MpsActivator0 sending TH2 to MpsActivator0 S[START-
ING,READY] p:9 t:-1.0
18     [0000.7000] MpsActivator0 TH2 to MpsActivator0 S[STARTING,READY]
p:9 t:-1.0 delivered
19     [0000.7000] MpsActivator0 EXECUTING CTXSW_TO_RUNNING1-action [RUN-
NING] P:9
20     [0000.7000] MpsActivator0 delayedDelivery 1.4
21     [0001.4000] MpsActivator0 TH3 to MpsActivator0
S[CTXSW_TO_RUNNING1,RUNNING] p:9 t:1.4 delivered
22     [0001.4000] MpsActivator0 EXECUTING GETTING_MSG-action [RUNNING]
P:10
23     [0001.4000] MpsActivator0 CHARGING 0.0016897916793823242
24     [0001.4000] MpsActivator0 delayedDelivery 1.4016897916793822
25     [0001.4017] MpsActivator0 TH3 to MpsActivator0 S[GETTING_MSG,RUN-
NING] p:10 t:1.4016897916793822 delivered
26     [0001.4017] MpsActivator0 EXECUTING ACTIVATE-action [RUNNING] P:10
27     [0001.4017] Plato WAITING: %g TK_A1(Plato) to Token_A
```

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
8725 John J. Kingman Road, Ste 0944
Ft. Belvoir, VA22060-6218

2. Dudley Knox Library 2
Naval Postgraduate School
411 Dyer Road
Monterey, CA 93943-5101

3. Dan Boger, Chairman, Code CS 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100

4. Dr. Tim Shimeall, Code CS/Sm 3
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100

5. Technical Library Branch 1 1
SPAWARSYSCEN D0274
53560 Hull Sreet
San Diego, CA 92152-5001

6. Larry Peterson (PL-SS) 1
SPAWARSYSCEN D44202
53140 Systems Street, Room 212
San Diego, CA 92152-7555

7. Roger Merk (PL-TS) 1
SPAWARSYSCEN D8203
49258 Mills Street, Room 158
San Diego, CA 92152-5371

8. Cynthia Keune 1
2149 Fairfield Sreet
San Diego, CA 92110

9. Robert M. Ollerton (PL-TS) 3
SPAWARSYSCEN D827
49258 Mills Street, Room 158
San Diego, CA 92152-5371