Parallel Simulation of Communicating Finite State Machines

*Carl Tropper** *and Azzedine Boukerche*†
Jet Propulsion Laboratories
California Institute of Technology

Abstract

We describe, in this paper, a synchronization/deadlock resolution mechanism for a network of communicating finite state machines implemented on a parallel machine. As it is message-based, it is appropriate for distributed memory machines.

The technique was inspired by a project at the Jet Propulsion Laboratories whose goal is the specification and verification of the software used to control the interplanetary spacecraft operated by the laboratory.

The network of communicating finite state machines makes use of write messages to alter the value of the variables describing the finite state machines and read messages to determine the state of the variables. Since a blocking protocol is employed, it is possible for deadlocks to occur. Consequently, we describe deadlock resolution algorithms.

Our algorithms were implemented on an iPSC/2 hypercube, demonstrating good performance on a queueing network model.

1 Introduction

We describe, in this paper, a synchronization mechanism for a network of communicating finite state machines implemented on a parallel machine. As it is message-based, it is appropriate for distributed memory machines.

The technique was inspired by a project at the Jet Propulsion Laboratories of the California Institute of Technology, whose goal is the specification and verification of the software used to control the interplanetary spacecraft which are operated by the laboratory.

In order to control the actions of an interplanetary spacecraft, a sequence of commands are issued from the ground to on-board flight computers in the course of the spacecraft's voyage (fire engines, move radar, begin plotting surface...). Before actually issuing the commands, they are subject to verification so that "undesirable" states are not entered into. For example, instruments which are susceptible to the sun's glare should not be unnecessarily exposed to it. At present, the verification consists of checking flight rules via a sequential event-logic language which was originally developed for the Voyager program in the mid-70's [ALKA91]. In an attempt to place the specification and verification of the spacecraft on a more modern foundation, a model of the spacecraft, based on the notion of communicating finite state machines (abbreviated henceforth as fsm's) was proposed in [ALKA91]. A transition between the states of an fsm occurs as a consequence of a command executed at the fsm. In appendix A we provide an example (based on [ALKA91]) of a small subsystem of the Galileo flight software in order to make this approach to the specification and verification of real-time systems more concrete.

Since the communicating fsm's are to be distributed among the nodes of a multi-computer, a synchronization technique is required in order to assure that the causality of the model is not violated. In the context of our model, causality means that the write events are simulated in chronological order at each fsm and the read events are simulated for the correct values of the variables which they are trying to access.

While *conservative* synchronization mechanisms rely on blocking to avoid violations of dependance constraints, *optimistic* methods rely on detecting synchronization errors at runtime and on recovery using a *rollback* mechanism. The algorithms we consider in this paper are an extension to a conservative mechanism. In particular, we present an approach to synchronizing the fsm's based upon the use of request messages. The technique is an outgrowth of algorithms for synchronizing parallel simulations as described in [COTE92] and [COTE92]. Algorithms for parallel simulation which are closely related to this approach are described in [MISR86], [FUJI90] and [NICO88]. Our work differs from these algorithms in that our algorithm supports the read operation of the state of variable in distinct nodes (fsm's in our case).

This technique is particularly important when the immense size of these models is taken into account. In order to be able to verify a model such as the one described above in a reasonable time it is necessary to execute it on a parallel machine. Hence the need for efficient synchronization techniques.

The remainder of this paper is organized along the following lines. We present our model for synchronizing the fsm's in the next section, discussing the ways in which deadlocks can arise. The third section contains a description of the algorithm, followed by a proof of its' correct-

*On sabatical leave from School of Computer Science, McGill Univ. Canada

†Visiting doctoral student at the California Institute of Technology

ness. In section 5 we describe the experiments and the performance of the algorithms. The conclusion follows.

2 The Model

We model the communicating fsm's as a directed graph in which the nodes are fsm's and the links represent queues of either write[1] or read messages. Messages are exchanged between the fsm's which are intended to represent the sending of write and read messages from one fsm to another and the sending of state information between fsm's. Write messages are used to write to state variables, while the messages which request the state of an fsm correspond to the reading of state variables. The messages in the queues await processing at the fsm's.

Write messages, denoted by $< Write, write_{time} >$ may change the value of a fsm's variables. $Write_{time}$ is the (simulated) time of the write message. Since our model is conservative, writes must be processed in chronological order and no processing can occur until either the missing write arrives or until an estimate is obtained of the earliest time at which a write message can arrive. In order to complete the (simulated) processing of a write message, it may be necessary for the fsm to request the state of a variable in another fsm (for example, in the case of verification program described in the appendix A, it may be necessary to first obtain the value of a variable located at another fsm in order to determine the safety of a transition). This is accomplished via read message sent to the fsm in question. The read message takes the form $< Read, read_{time} >$, where $read_{time}$ is the (simulated) time at which the read is to occur.

Until a reply is received, the processing of the write message cannot be completed, and no other write message can be processed at the fsm. Read messages can be processed, but only up to the time at which the last write occurred. Since no further writes can be processed, read messages with times larger than the time of the last update of the variables which they are trying to read cannot be processed. We say that the fsm in question is (temporarily) blocked.

It is assumed that there is no upper bound on the amount of time it takes a message to traverse a link, there is no loss of messages, links never fail and they are FIFO (do not reorder or duplicate messages). Furthermore, we assume that the connectivity of the graph never changes.

In our model each fsm is represented by a logical process LP[2]. The LP maintains a local simulation time lt, defined as the last time at which a write was executed at the fsm. Separate queues are maintained for the read and the write messages.

Each of the LPs is initialized with a queue of writes. (In our example, these can be the set of writes which are initially sent to the spacecraft or can be the result of the pre-simulation phase proposed in [ALKA91]).

[1] In the sequel write/read messages and write/read-event messages are taken to mean the same things

[2] In the sequel LP is to taken to mean the logical process that will simulate the fsm in our model.

As mentioned above, an fsm will block when it has an empty (write) input queue. In order to obtain the missing write message, our LP sends a $< Request_{write}, t_{smin} >$ message (t_{smin} is the minimum time stamp at the LP) to the appropriate neighbor. In the event that the neighbor can provide a write message for the requesting LP, it does so. If it does not have one, then it examines its' local time *(lt)* to see if it is larger than t_{smin}. If $lt > t_{smin}$, then it sends a $< reply, lt >$ message back to the requesting LP. When this message arrives, it unblocks the LP. If $lt < t_{smin}$, and if the fsm has empty input queues itself, it sends its' own $< Request_{write}, lt >$ message to these neighbors which correspond to its' empty queues.

Unfortunately, blocking behavior can lead to deadlock. We turn to a description of several of these deadlocks.

2.1 Different Type of Deadlocks

Deadlocks can occur during simulation as a result of blocking and limited buffers. Since we employ different deadlock-breaking mechanisms depending upon the type of deadlocks which occurs, we analyse different cases. We distinguish between the following types of deadlocks:

(1) Write Deadlock

Our first example of deadlock is due to empty (write) input queues. Consider three LPs, depicted below in figure 1. Each of the LPs has an empty input queue, expecting messages from another LP. The arrows indicate this dependency. At the same time, each of the LPs have writes in other input queues. LP_1 has a write bearing time stamp 11, LP_2 has a write with time stamp 7, and LP_3 has one bearing time-stamp 13. The local time at each LP are shown inside the circle. (The local time is the time at which a write was last executed).

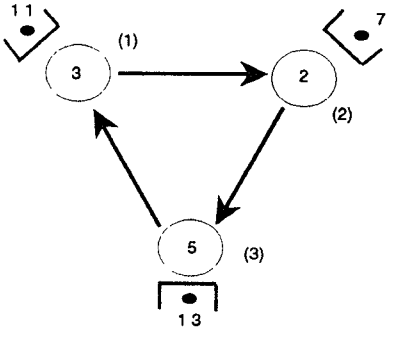


Figure 1: **Cycle of Request Writes**

In accordance with our protocols, LP 1 requests a write from LP 2, LP 2 does the same thing to LP 3 and LP 3 completes the circle. None of the LP's can respond because they cannot be certain that they won't receive a message from their neighbor which could change its' response.

Our deadlock is, of course, a cycle. It readily generalizes to a knot.

Definition of a Knot 2.1 *Let (V,E) be a directed graph where* V *is the set of vertices in the graph and* E *a set of directed edges where a directed edge* (i,j) *indicates that in node* i *the link* (i,j) *is designated as an output link.*

A knot K *in* (V,E) *is a strongly connected subgraph of* (V,E) *with no edges directed away from the subgraph* K. *Alternatively, a node* i *is a member of a knot* K *if node* i *is reachable from all nodes which are reachable from node* i. *Each of these nodes must be able to reach* i.

(2) Read Write Deadlock

A deadlock may also involve both read messages as well as request for write messages. As illustrated in figure 2, we can see in the example that LP_B cannot respond to the read message $< Read, 5 >$ issued by LP_A because it has an empty input queue, and it local time $lt_B(= 3)$ is less than 5. The arrows from LP_B to LP_C and LP_C to LP_A indicate an empty input queues at LP_B and LP_C respectively. LP_A is blocked until it completes processing the read message.

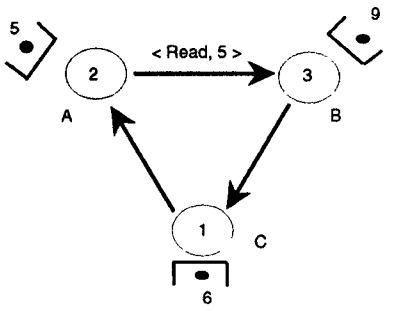


Figure 2: **Read-Write Deadlock**

The preceding two examples illustrate the occurrence of deadlocks as a consequence of awaiting messages (which don't arrive).

(3) Memory Deadlock

Lack of memory can also cause deadlocks [COTE92]. If no upper bound is placed on the number of buffers associated with each LP, then dynamic memory management could significantly reduce the possibility of deadlock. In what follows, we assume that there is sufficient memory for these deadlocks not to occur.

3 Description of the Algorithm

We start with an informal description of the algorithm, and then make it more precise.

Let us first assume that the deadlock takes the form of a cycle, i.e. that 2 LPs "connected" by the read message in our dependency graph are also connected by a path of request write messages, as depicted in the example above (fig 2). An arrow directed from LP_B to LP_C in a dependency graph means that LP_B has an empty input queue from LP_C.

Recall that a read message may be issued by an LP while processing a write message.

The read message is defined by:

$$< Read, read_{time} >$$

where $read_{time}$ is the (simulated) time of the read message issued by an LP to read the value of a variable located in another LP.

Denote by $LP_{id_{launch}}$ the LP which issues the read message and by $LP_{proc_{init}}$ the LP which receives the read message.

In general, a read message may be responded to if and only if $read_{time} < max(lt(proc_{init}), t_{smin})$, where t_{smin} is the smallest time stamp in all of the $LP_{proc_{init}}$'s input queues, the $read_{time}$ is the read time of the read message issued by $LP_{id_{launch}}$ and the $lt(proc_{init})$ is the local (clock) time at the $LP_{proc_{init}}$.

Upon receipt of a read message $< Read, read_{time} >$ from $LP_{id_{launch}}$ (LP_A in figure 2), $LP_{proc_{init}}$ (LP_B in figure 2) sends a $request_{bad}$ (request blocked_awaiting_data) message along its' empty input queues, i.e. to neighboring LPs which deliver messages via the empty queues. The purpose of this request message is to determine if the value of the variable requested by the read message at $LP_{proc_{init}}$ will change between the $lt(id_{proc})$ and $read_{time}$. The $request_{bad}$ message is in turn forwarded along a path of empty input queues to $LP_{id_{launch}}$. If the $request_{bad}$ message arrives at $LP_{id_{launch}}$ then $LP_{id_{launch}}$ checks to see if it has received the data which it requested by the read message. If it has not, we have a read-write type of deadlock (see figure 2). Therefore $LP_{id_{launch}}$ sends a bad (blocked awaiting data) message which returns to $LP_{proc_{init}}$ via the same path taken by the $request_{bad}$ message (but in the oppposite direction). In our example, the $request_{bad}$ message travels from LP_B to LP_A via the empty links, and the bad message returns in the opposite direction. Upon receipt of the bad message, $LP_{proc_{init}}$ is free to either respond to the read message or to process a write message in one of its' input queues.

We now turn to a more precise description of our algorithm.

As described in section 2, we make use of $< Request_{write}, t_{smin} >$ messages in an attempt to unblock LPs which have empty (write) input queues.

In the event that the following the 2 conditions are satisfied, the LP initiates the knot detection algorithm described in [MISR83], in order to detect a write deadlock.

1. The LP must have received at least one request and have sent one request. This condition eliminates those LPs which could not possibly belong to a knot from starting the knot detection algorithm.
2. The time associated with the request write message issued by the LP must be smaller than or equal to the time of some request write message which was received by the LP. The purpose of this condition is to exclude those LPs which cannot break the knot from initiating the detection algorithm.

The deadlock is then broken by detecting the LP with the smallest time stamps among the LPs involved in the deadlock [BOUK90].

Recall that $LP_{id_{launch}}$ is the LP which issues the read message and $LP_{proc_{init}}$ is the LP which receives the read message.

Upon receipt of a read message, $LP_{proc_{init}}$ sends a $request_{bad}$ message along its' empty input links if it is unable to reply.

The $request_{bad}$ message is defined by:

$$< Request_{bad}, read_{time}, id_{launch}, proc_{init} >$$

where $read_{time}$ is the read time of the read message issued by $LP_{id_{launch}}$, id_{launch} is the id of the LP which sends the read message, $proc_{init}$ is the id of the LP which sends the $request_{bad}$ message after receiving a read message

Each LP maintains a data structure so that it may forward only one $request_{bad}$ message corresponding to the same read message along its' outgoing links. The data structure is

$parent_{id_k} = id$ of the LP from which a $request_{bad}$ message arrives, such that k is the id of $LP_{id_{launch}}$. In our example at LP_C, $parent_A = LP_B$.

$parent_{rt_k}$ contains the associated read time; in our example, $parent_{rt_A} = 5$.

Upon receipt of a $request_{bad}$ message, both fields are examined, i.e $parent_{id_k}$ and $parent_{rt_k}$. If the message comes from a different parent and has the same read time, then it is a duplicate of one already received and is discarded, thereby reducing the number of $request_{bad}$ messages involved in the simulation.

When a request bad message arrives at an LP, then

1. If the LP is not blocked, it keeps the message untill such time as it blocks.
2. If the LP is blocked and the $read_{time} \leq lt$ then the request bad message is forwarded to neighbors which correspond to empty (write) input links. If the $read_{time} > lt$, a bad message is sent along the path determined by the parent_id's. We describe the bad message and its' processing below.

If a $request_{bad}$ message finally arrives at $LP_{id_{launch}}$, then if it is blocked waiting for data, a comparison of the $read_{time}$ associated with the request bad message and the lt of $LP_{id_{launch}}$ is made. If they are equal, a bad message is sent to the LP from which the request bad message arrived. Subsequent $request_{bad}$ messages corresponding to the same read are discarded. The bad message is defined as

$$< Bad, id_{launch}, proc_{init}, read_{time} > .$$

The bad message now returns along the path dictated by the *parent_id* 's. When a a bad message arrives at an empty write input queue, it is processed in the order dictated by its' time stamp.

In the event that the bad message reaches $LP_{proc_{init}}$ (LP_B in our example) its' $read_{time}$ must fall into one of two cases:

1. $lt_B < read_{time} < t_{smin}(B)$. Then LP_B may respond to LP_A's $< Read, read_{time} >$ message by sending the value of the variable requested via a data message.
2. $lt_B < t_{smin}(B) \leq read_{time}$. Then the write at LP_B may be executed.
   As before it is still possible that execution of the write might not result in the read being responded to; a write bearing a time stamp >= the time of the read might have to first arrive.

Recall, that if $LP_{proc_{init}}$ unblocks, it may provide that data message to the read message.

The *data* message is defined by:

$$< Data, data_{time}, id_{launch}, proc_{init} >$$

where $data_{time}$ is the data time of the recent value of the variables requested by the read message issued by $LP_{id_{launch}}$, id_{launch} is the id of the LP which sends the read message,

4 Proof of Correctness

To simplify our discussion, call a deadlock which only involves read messages a read-only deadlock, one which only involves write messages a write-only deadlock, and the combination a read-write deadlock.

We start out with a

Lemma 4.1 *Read-only deadlocks cannot occur.*

Proof

We assume that the response to a $< Read, read\_time >$ message is the state of a variable at time $t - \epsilon(\epsilon > 0)$. We make this assumption in order to maintain temporal consistency, as the write operation which resulted in the $< Read, read_{time} >$ message should be executed after it obtains the necessary state information. This assumption also avoids deadlocks, as we shall see .

We show that in order for such a deadlock to occur, the time-stamps of the corresponding write messages (i.e. which resulted in the read messages being issued) must be equal. As a consequence of our assumption, it follows that all of the LPs can respond with a $< Data, read_{time} - \epsilon >$ message.

Assume that there exists a deadlock and that not all of the minimum time-stamps of the read messages are equal. Then a $< Read, read_{time} >$ must arrive at some LPi such that $read_{time} < t_{smin}(LPi)$. LPi will then be able to

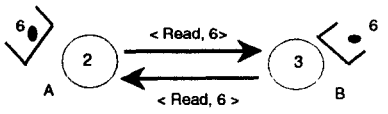


Figure 3: **Read Cycle**

respond with a data message, contradicting the existence of a deadlock. □

Lemma 4.2 *Write only deadlocks can be detected and broken.*

Proof

Write only messages may form a knot, as pointed out in the preceding section. We may detect this knot by use of the algorithm described in 3. In order to be able to break the knot, we also detect the LP which sent the request with the smallest request time. The LP with the smallest request time can respond to at least one request, thereby breaking the knot. □

Having eliminated these two categories of deadlocks, we are left with the possibility of a deadlock which involves both read and request write messages. In the spirit of divide and conquer, we first prove

Theorem 4.1 *A deadlock which involves a single read message and more than one write message may be broken.*

Proof

The reason that we restrict ourselves to deadlocks involving one read and more than one write message is that a deadlock involving exactly one read and one write message cannot occur. Such a deadlock must, per force, involve 2 LPs, as illustrated below.

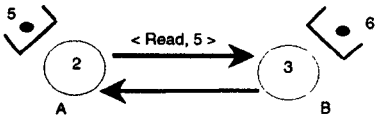


Figure 4: **Read-Write Cycle**

LP A is constrained to execute the write bearing its' smallest time stamp. Consequently, upon receiving the read message, LP B is aware that it cannot receive a message bearing a smaller time stamp the time of the read message. Hence there is no deadlock. (We note, however, that LP B may not be able to immediately respond to the read message because its' own minimum time stamp is less than the time of the request. In this case it executes the write with the smallest time stamp and continues processing writes until its' minimum time stamp exceeds the time of the read message). □

Summarizing the argument to this point, we may say

Lemma 4.3 *If a deadlock cycle exists, a* $request_{bad}$ *message is sent and arrives at* $LP_{id_{launch}}$. *In turn, a bad message issued by* $LP_{id_{launch}}$ *will arrive at* $LP_{proc_{init}}$, *thereby breaking the deadlock.*

Proof

The final case which we must dispose of in order to prove the theorem is the existence of a knot of write messages in between the LPs connected by the read message. An example of this is depicted in the figure below.

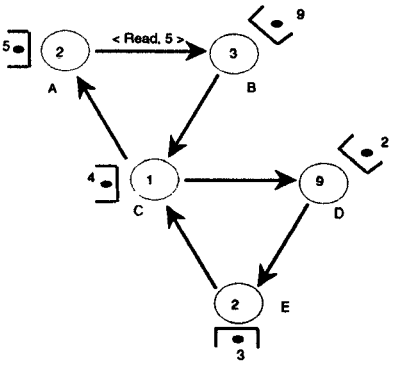


Figure 5: **Read-Write Knot**

In this example, the arrival of a bad message at node C will not unblock C. However, the knot detection/breaking algorithm previously described will locate the minimum (write) time stamp in the knot. Since no write message with a smaller time stamp can arrive in its' input queues, the LP which has the write bearing the smallest time stamp may be processed. In the event that is LP is blocked awaiting a read, it can demand the data via a $< Demand_{read}, t >$ message. □

We are now in a position to construct the final link in our exclusionary chain.

Theorem 4.2 *A deadlock which involves more than one read and writes cannot occur.*

Proof

Consider an LP which has received a read message, but which cannot respond to the to the read because of an empty input queue. The empty input queue is directed to an LP which did not send the read message.

The minimum time stamp at the LP must be > the smallest time stamp of any write message which can arrive in the empty queue. This is true because a $request_{write}$ message would have produced an estimate which would have unblocked the LP. (the write bearing the smallest time stamp would have been executed). Furthermore, $t_{smin}(write) < ts$ (read), else the read message could complete.

The subsequent LP must issue either a read or a request write message. Since the LP is part of a knot, a path of LPs which issue either reads or request writes must lead to the the LP which sent the original read message. Making use of the above argument, a simple inductive argument leads to a strictly decreasing sequence of time stamps associated with the read messages. This leads to a contradiction. □

Summarizing the above discussion, we have

Theorem 4.3 *Deadlocks involving write messages alone may be detected and broken, while deadlocks involving read and write messages may either be prevented or detected and broken.*

5 Performance- Experiments and Results

In order to investigate the performance of our algorithms we elected to simulate them on an Intel iPSC/2 hypercube.

The iPSC/2 is a distributed memory multiprocessor, in which the processing elements are connected in a hypercube topology. The iPSC/2 consists of nodes and a front-end processor. Each node is a processor/memory pair and runs the NX/2 operating system and uses message passing to other nodes. The iPSC/2 board consists of an intel 80386 processor and a 80387 co-processor, both running at 16 MHz, local memory of up to 4 MBytes, and a 32 bit architecture.

The time T required to transmit a one hop message of length N bytes is : $T = \alpha + \beta N$

where α represents a fixed start-up time (= 390 microseconds) for messages less than 100 bytes and β represents the transmission time per byte (=0.4 microseconds) (see [DUNI90] for a description of the performance of the hypercube).

We chose the graph of our network of communicating LP's to be a torus because the large number of cycles in the torus provides a stress test for the algorithms. The presence of cycles results in the knot detection algorithm being frequently called if the number of messages present in the graph is sufficiently small. In keeping with this approach, we initialized each link of the torus with 3 write messages. We can interpret the initial messages as the first set of messages sent to the spacecraft. We made use of several size torii, from dimension 6x6 to 14x14 and varied the number of processors from 2 to 16 processors.

We made use of a simple static mapping strategy of LPs to processors, as in [FUJI90], [BOUK90] in which a torus is subdivided into grids, and in which the LPs in the same grids are allocated to the same processor.

The simulation runs for 1000 units of simulated time, where one time unit is taken to be the time required to process a message. All categories of messages are assumed to require one unit of processing time. We assume that as a result of processing a write message at most one read message and at most one write message can be generated. A write is generated with probability 80% while a read is produced with probability 20%. Once produced, the probability that a message is directed to a neighboring link is determined by a uniform distribution, i.e. each output link is equally likely to be chosen. We present our results below in the form of a graph of the execution time (in seconds) of the model as a function of the number of processors employed to execute the model. As we can see from the curves (figures 6, 7), the algorithms exhibit good performance, in which the run time decreases significantly with an increase in the number of processors.

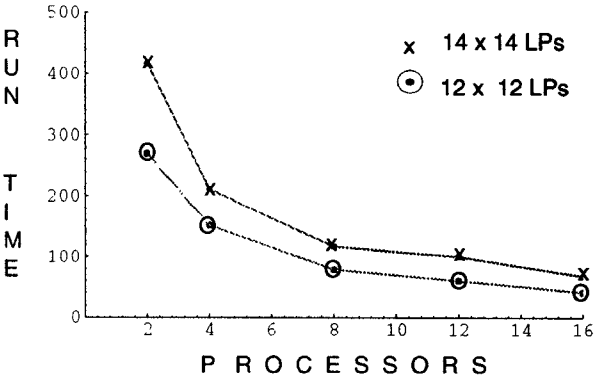


Figure 6: Performance - Run Time vs. Processors

From the shape of the graphs we note that the largest decrease in execution time occurs initially, with the run time eventually flattening out. For example, in figure 6 increasing the number of processors from 2 to 4 results in approximately a 50% decrease in run time. From 4 to 8 processors, we observe a 30% decrease. After 8 processors the curve gradually flattens out. A number of factors contribute to the shape of the curve.

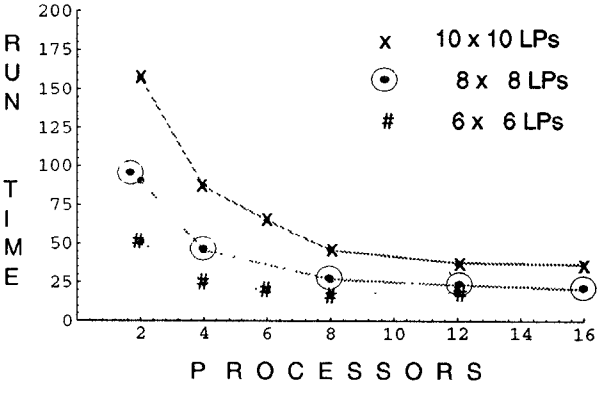


Figure 7: Performance - Run Time vs. Processors

First is the limit on the parallelism of the model. The number of LPs in the model coupled with the number of messages initially placed at each of the LPs determine the amount of parallel activity in the model. While the shape of the graphs in figures 6 and 7 are the same, the percentage decrease in the run time of the models is smaller when there are fewer LPs.

A second factor is the effect of inter-processor commu-

nication, in particular the increase in the cost of knot detection and breaking when message passing is involved. The knot detection algorithm used is the one described in [MISR83].

6 Conclusion

We described, in this paper, algorithms for synchronizing a collection of communicating finite state machines and for resolving deadlocks which might arise in the process. The algorithms were implemented on an Intel iPSC/2 and were tested in the context of a queueing model of a torus. The torus was chosen primarily because it is rich in cycles and is therefore a stress test for our algorithms.

The performance results show that significant decreases in run time were obtained with the use of the algorithms described in this paper.

An issue worthy of further attention is that of memory management. It becomes more important as the program grows in size, especially the management of space for write messages.

Our approach to memory management has been to employ a fixed number of buffers at each LP. However, as pointed out in [COTE92], deadlocks may result. One approach to their resolution is described in [COTE92]. Another approach is to use a dynamic scheme in which all buffers are shared by LPs, subject to an upper and a lower bound on number of buffers available to an LP. This prevents starvation and hogging. A combination of a process migration protocol with the dynamic scheme might also be envisaged.

The algorithms outlined here are suitable for addressing related problems in domains other than the verification of real-time software programs. Examples are combat simulation and shark world simulation, to mention a few, where it is convenient to utilize state variables that can be accessed by distinct logical processes.

Appendix A

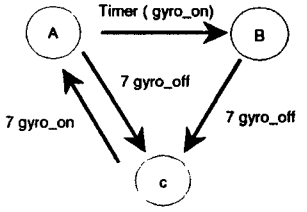
We provide an example (based on [ALKA91]) of a small sub-system of the Galileo flight software.

Our example is based upon a small sub-system of the Galileo spacecraft. It illustrates the use of communicating finite state machines for the purpose of specifying and verifying a rather large, complicated real-time system. Subsystems of the spacecraft are represented by fsm's, with safe and un-safe state transitions specified. Messages are exchanged between the fsm's which are intended to represent the sending of commands from one fsm to another and the sending of state information between fsm's. The commands are represented by messages which write to state variables, while the messages which request the state of an fsm correspond to the reading of state variables.

The system consists of a gyroscope, an inertial sensor, an accelerometer and a heater. The flight mode of the spacecraft is maintained by an attitude and articulation command subsystem, while a command data system receives command subsequences which are transmitted to the spacecraft from the ground and schedules them for execution.

	7gyrop_on	7gyrop_off	Timer
S0	S 0 Schedule (71N1P,t1) Else {Schedule (71N1PR,t1) Schedule (71N1PR,t1)}	Error	Error
S1	S 1 Error	Schedule (71N1R,t1)	Error
S2	S 2 Error	If not (ins_on) Schedule (71N1R,t1)	Error

Table 1: **State Actions.**



State { (X,Y) | X = gyro_power, Y = reliable_dat }

Figure 8: **State Graph**

The gyro is powered on in response to a *7gyro_on* command, and is powered off upon receipt of the *7gyro_off* command. It produces reliable data only after the inertial sensor has been turned on for 40 seconds after the gyro is powered on. The gyro turns on the inertial sensor by scheduling the *71n1p* command and turns it off by issuing the *71n1pr* command. 40 seconds after the inertial sensor is turned on, it notifies the gyro that its' data is reliable by issuing the timer(gyro) command. At point the gyro is considered to be functioning reliably.

The accelerometer functions in a similar fashion. When it is turned on via the *7accl_on* command, it also requires that the inertial sensor be switched on. If the gyro has previously turned on the inertial sensor, the accelerometer does not have to do so. When the gyro is powered on after the accelerometer has already turned on the inertial sensor, the gyro will first switch off the sensor and then turn it back on. This is done because the gyro requires that the inertial sensor be initialized.

The gyro heater is switched on by the *7htr_on* command and off by the *7htr_off* command.

Figure 8 contains a finite state machine for the gyro, Fsm's for the inertial sensor, and the heater may be found in [ALKA91]. Each state is represented by two variables, indicating whether the power is turned on and whether the data is reliable. A transition between the states of an fsm occurs as a consequence of a command executed at the fsm. An altered state can, in turn, cause the generation of new commands. An example of such "state actions" is contained in the table 1 below devoted to gyro actions.

Several entries in the table indicate that an error state has been reached. In order to determine whether such a

state occurs, constraints on allowable transitions are examined after the execution of each command. These constraints, based in part upon the Galileo flight manual, are described by the following pseudo-code:

1. whenever ($gyro\_on < -- true$)
   ($gyro\_on.t - htr.on.t < 1$ hour and $htr\_on.v = true$) <=> error
2. whenever ($gyro\_on < -- false$)
   ($aacs\_mode = inertial$) <=> error
3. $aacs\_mode.t - gyro\_on.t < 2$ hours <=> error

Acknowledgments

Heartfelt thanks to Mani Chandy for arranging CT's sabbatical at JPL and for reviewing the manuscript and providing criticism. Thanks to John Horvath at JPL for hosting CT's visit. Thanks are in order to Richard Fagen, director of the Campus Computing Organization of the California Institute of Technology for providing us with computational facilities and to Paul Messina, director of the CCSF at CalTech for providing us with super-computer access.

References

[ALKA91] Alkalaj L., "Towards a Specification Language and Programming Environment for Concurrent Constraint Validation of Spacecraft Commands", JPL, Pasadena, Calif. 1991.

[BOUK90] Boukerche A., " Performance Analysis of Distributed Simulation", M.Sc. thesis, McGill Univ., School of Computer Science, Montreal, Canada, 1990.

[COTE91] Cote C., "Pseudosimulation in Distributed Simulation", M.Sc. thesis, McGill Univ., School of Computer Science, Montreal, Canada, 1991

[COTE92] Cote C. and Tropper C., "On Distributed and Pseudosimulation", 1992 Workshop on Parallel and Distributed Simulation, SCS, Vol. 24, n. 3, Jan 1992, pp. 97-106.

[DUNI90] Dunigan T. H., "Performance of the Intel iPCS/2 Hypercube", Technical Report ORNL/TM-11491, 1990, Oak Ridge National Laboratory, Oak Ridge, TN, 37831

[FUJI90] Fujimoto R., " Parrallel Discrete Event Simulation", Communication of the ACM, October 1990.

[MISR83] Misra J. and Chandy K. M., "A Distributed Graph Algorithm:Knot Detection", ACM Transactions on Programming Languages and Systems, vol. 4, no. 4, Oct. 1983, pp. 678-686

[MISR86] Misra, J. , "Distributed Discrete Event Simulation", ACM Computing Surveys, 18 (1), March, 1986, pp 39-65

[NICO88] Nicol, D., "Parallel Discrete Event Simulation of FCFS Stochastic Queueing Networks", Proc. ACM SIGPLAN Symposium on Parallel Programming Environments, Applications, and Languages, Yale University, July, 1988

[TROP92] Tropper C., Boukerche A., "Parallelizing the Sequencing Problem", TR-SOCS, 1992, McGill University, Montreal, Canada