Reducing Null Messages in Misra's Distributed Discrete Event Simulation Method

RONALD C. DE VRIES, SENIOR MEMBER, IEEE

***Abstract*—The paper considers the implementation of distributed discrete event simulation (DDES) using what has been commonly called the Misra approach, after one of the inventors of the approach. A major problem with DDES is that deadlock can occur. Therefore, DDES algorithms must either avoid deadlock in the first place, or detect the existence of deadlock when it does occur and eliminate the deadlock. Misra proposes the use of NULL messages as one way to circumvent the deadlock problem. However, the number of NULL messages can become quite large. Methods are presented for reducing the number of NULL messages through the prediction of channel times. A framework is presented on which the distributed discrete event simulation can be built for applications which can be decomposed into feedforward and feedback networks.**

***Index Terms*—Distributed discrete event simulation.**

I. Introduction

DISTRIBUTED Discrete Event Simulation (DDES) has gained considerable interest because of applications which do not fit into the category of fixed time-step operation, but which are large enough problems that distributing the problem over a number of computers is attractive from the point of view of computational time. One basic problem that must be solved with DDES is the avoidance or elimination of deadlock. Two basic methods have been proposed for implementing DDES.

The first method, proposed by Jefferson and referred to as the Time Warp method [1], accepts and processes messages when they are received even though doing so could cause the message to be incorrectly processed out of order. If and when a message is detected as being processed out of order, rollback is performed. That is, time is rolled back to the point where the erroneous processing occurred, and is restarted with processing taking place in the correct order. Messages sent erroneously must be canceled by "antimessages."

A second basic approach is that proposed by Chandy and Misra [2] and Misra [3]. The same problem was independently studied by Peacock, Wong, and Manning [4]. Jayaraman [5] applies data flow concepts to the DDES problem and Kumar [6] specifically investigates feedforward systems using distributed processors. The approach taken in this paper is based largely on [3].

Manuscript received March 9, 1989; revised August 22, 1989. Recommended by P. A. Ng. This work was supported by Los Alamos National Laboratories under Contract BOA 9-X66-W2078-1, Task 18.

The author is with the Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, NM 87131.

IEEE Log Number 8931763.

One suggestion made by Misra [3] to circumvent the deadlock problem was to use NULL message. NULL messages are messages that contain no real message content and have no analog in the real system being simulated; they are simply used to advance time within the system. When the possibility of deadlock is detected, a NULL message is sent in the hopes that time will be sufficiently incremented to allow real messages to proceed thorough the system. The number of NULL messages can be quite extensive.

A framework is presented here on which DDES can be built for applications which can be decomposed into feedforward and feedback networks. Then ways are presented for reducing the number of NULL messages that are required in Misra's approach. The starting point is a system that has been designed initially using series, feedforward, and feedback networks or has been decomposed as fully as possible into such networks. Research is presently being performed into determining a procedure for starting from a general flow diagram of the total system and performing the decomposition operation mentioned above.

In the following, the reader is assumed to be familiar with the message passing concept and Misra's method for DDES in particular.

II. Definitions and Assumptions

Definitions for some terms that are used in the paper are presented next.

Simulation time is the time kept by a process or channel that specifies the time in the real system that is being simulated. The simulation time kept by a process is called the *local* time.

The simulation time at which the process starts processing the message is called the *initiation time*, and the simulation time at which it completes processing the message and is able to accept another message is called the *next initiation time*.

A *timestamp* is a time associated with a message that states the simulation time at which the receiving process is to receive the message. It is also the local time at which the previous process sent the message.

A *merge* element is a node at which a process is implemented, and which has two or more input channels and one output channel. Whenever it accepts a message on one of its input channels, it sends a possibly modified message on its output channel. A merge element acts like a First Come, First Serve (FCFS) queue. An FCFS queue is also called a First In, First Out (FIFO) queue.

A *disperse* element is a node at which a process is implemented, and which has two or more output channels and one input channel. Whenever it accepts a message on its input channel, it sends a possibly modified message on one of its output channels.

A number of assumptions need to be made. These are listed below:

1) Unless stated otherwise, a process produces exactly one output message in response to an input message. The message that is produced could be a NULL message.

2) Messages arrive in timestamp order along any channel between two processes. They need not arrive in timestamp order if they arrive on different channels.

3) A process is capable of processing only one message at a time. When it processes a non-null message, it determines its new local time by taking its present local time and adding to it the delay that would be introduced by the real system performing the operation defined by the message.

4) Local time is determined in the following way. If local time is earlier than any channel time, then local time is made equal to the lowest channel time. If the local time is later than any channel time, the local time is retained until the processing of a message takes place. The new local time is the present local time plus the simulation time associated with the message just processed.

5) A NULL message serves only to update times. It is handled in the same way as a normal message except that the minimum simulated delay is added to the local time to determine the timestamp of the NULL message that is sent. However, the local time is NOT changed by the processing of this NULL message, since no simulation occurred.

6) The send time of a message is also the receive time of the message at the next process.

7) In order to simplify the algorithms somewhat, the assumption is made that a termination message sets the channel time to infinity and that the channel time is NOT changed thereafter. When all channels read infinity, the processing stops. The termination aspect is not stated explicitly in any of the algorithms presented here.

The reason that local time is updated in the fashion indicated in Assumption 4 may not be clear. Normally the local time will be that of the lowest time over the set of channel times. However, the local time of a process, such as a queue, may be a later time. The local time will be the time at which the process accepts the message and starts processing it. For example, in the case of a cashier, the message would contain the time at which the customer arrives at the queue. The channel takes on this receive time. The local time of the cashier will be the time at which the cashier starts servicing the customer. Some additional time will then be added to account for the time taken by the cashier to serve the customer. The time associated with the exit message is the time the customer actually leaves the cashier. The requirement that the send and receive times be the same is made in order to assure that messages do not arrive out of timestamp order at the receiving end. If the send time is allowed to be different than the receive time, a message that is sent later than another could be received earlier and be out of timestamp order.

Misra's basic method is used to choose the next message to process. In Misra's method a message must be present on every channel. Then the earliest message on the channel having the lowest channel time is the message that is processed. If some channel does not have a message, the process must wait for messages to arrive on all channels before it can proceed. In the approach taken here, if a channel does not have a message, a prediction is made, where possible, on the earliest time that the next message can arrive over that channel. This predicted time is used as the channel time. Only if the channel having the lowest channel time does not have a message must the process wait for messages. The effect of using this method of choosing the next message to process at a merge element is to create a FCFS queue at that element. The number of NULL messages is reduced because, using the prediction, processes can proceed where Misra's basic method would not allow them to do so.

III. Prediction in a General Network

Suppose that the network can be only partly decomposed or not decomposed at all into subnetworks. Then steps can still be taken to reduce the number of NULL messages that must be sent. First, each successor process is given the minimum simulated delay associated with each of its predecessor processes. At initialization, each channel is initialized to 0. Therefore, a prediction can be made on the earliest time that a message can be received over that channel, which is the present channel time of 0 plus the minimum simulated delay for the predecessor process on that channel.

The time must remain the same until a message is received. A prediction can then be made as to the earliest time that the next message can be received over the channel, which is the local time of the predecessor process plus the minimum simulated delay. However, the local time of the process is only known at the time the last message was sent. One of the assumptions is that the new local time is the sending time of the message, which is also the receive time of the message. Therefore, the predicted time for the next message over the channel is simply the receive time plus the minimum simulated delay of the predecessor process. After processing the message, the channel time of that channel takes on this predicted time. This time is not changed until a new message is received over that channel or a prediction can be made based on other criteria.

Since the above prediction can be made at initialization and after the receipt of every real message, the assumption will be made that this prediction is actually performed. The execution of this prediction is formalized in the following assumption, Number 8.

8) At the time of initialization of the process, the time for each channel is updated to be that of the minimum simulated delay of the predecessor process for that channel. After every real (non-null) message is accepted and processed, the time of the channel on which the message was received is updated to be the present channel time plus the minimum simulated delay of the predecessor process for that channel. This procedure is assumed to be performed and is not explicitly stated in any of the algorithms that follow.

Some basic structures for implementing DDES are considered next.

IV. Simple Feedforward Network

In the following, a node of a flow diagram will be represented as N_x, where x is the node number. Node N_x is assumed to execute process P_x. In some cases, one may be emphasized over the other.

A *simple feedforward network* from node N_{f1} to node N_{f2} consists of the nodes N_{f1} and N_{f2} and two or more series paths from N_{f1} to N_{f2}. A series path from node N_{f1} to another node N_{f2} consists of a sequence of nodes starting from N_{f1} and going to N_{f2} with the direction of flow always from N_{f1} to N_{f2}. Consider a simple feedforward network having two such series paths, p_a and p_b. Let the simulation time at which P_{f1} completes processing of a message be T_n, the minimum simulated delay along p_a be t_a, and that along p_b be t_b. T_n was defined above as the next initiation time. As noted above, with elements such as queues, the receive time of a message, or timestamp, may not be the time at which the message is processed. The delay along p_a includes that of P_{1f} as does p_b. These minimum simulated times are the sums of the minimum simulated delays which are incurred by each process along each path. P_{f2} will receive the times t_a and t_b during initialization. Let P_{f1} send the following with any message along either path.

1) T_n.

2) An ascension number.

3) A tag which indicates that items 1 and 2 are to be passed along until N_{f2} is reached.

4) The time at which the next process along the path is to receive the message (the timestamp).

5) The message itself.

Each process along either path duplicates items 1, 2, and 3 above and replaces the timestamp and message with its own.

Let P_{f2} receive a message along p_a with next initiation time T_{na}. Then the earliest that another message can arrive along p_a is the larger of the present channel time and $T_{na} + t_a$. In addition, if we know that all messages have been received up to and including this message, then the earliest time that a message can be received along channel p_b is the larger of the present channel time of p_b and $T_{na} + t_b$. A similar statement can be made for p_b.

The above ideas can be used to define algorithms for the disperse and merge nodes. First consider P_{f1}, where P_{f1} has more than one input.

Algorithm Defining the Actions Taken by P_{f1} for a Simple Feedforward Network

1) Set ASCENSION = 1.
2) Set the time of each input channel to the lowest timestamp over all the messages waiting on that channel. If the channel having the earliest time has a message, process the message; otherwise, wait for a message to arrive on this channel and repeat this step.
3) Assign the number ASCENSION as the ascension number of the message to be output and send the message over the channel as determined by the process. Increment ASCENSION and go to Step 2.

The purpose of the ascension numbers is to allow P_{f2} to determine if any messages, sent by P_{f1}, up through the latest message have not yet been received.

Algorithm Defining the Actions Taken by P_{f2} for a Simple Feedforward Network

1) If a message has arrived whose ascension number is *not* the next one in order, retain it, wait for the next message, and repeat this step. Otherwise, continue placing received messages in their proper queue as long as the ascension number is the next in sequence. Then go to Step 2.
2) For each channel, determine the message in its queue, if one exists, with the lowest timestamp and assign that timestamp to the channel. If no message exists in the queue for channel x, assign the time $T_{ny} + t_x$ to p_x, where T_{ny} is the next initiation time in the message having the *highest* next initiation time of any message in a queue, unless the predicted time is less than that already on the channel. Make similar predictions for all the other channels which do not have messages in their queues. Update local time. If the channel with the earliest time does not have a message, go to Step 1; otherwise, go to Step 3.
3) Process the message and go to Step 2.

Note that ascension numbers die at merge points in simple feedforward networks.

Since no claim is being made that NULL messages can be completely eliminated, the only concern is that messages be processed in the correct order. All intermediate nodes along the paths have only one input channel. Therefore, Assumption 2 guarantees that these nodes will process messages in correct order. P_{f1} processes messages in the usual way and, therefore, in correct order. P_{f2} is confined to looking only at messages having an unbroken sequence of ascension numbers. Along each channel, the message with the lowest ascension number also has the lowest timestamp and the lowest next initiation time; otherwise, the requirement that messages along a channel be in timestamp order would be violated. Independent of the channel, the message with the lowest ascension number also has the lowest next initiation time, due to the way ascension numbers are assigned. Therefore, if all messages up to a certain ascension number are present and messages are selected according to Step 2, the channel predictions are correct and messages are processed in correct order. Consequently, the simple feedforward network processes messages in the correct order.

The above algorithm states that "if no message exists in the queue for channel x, assign the time $T_{ny} + t_x$ to p_x, where T_{ny} is the next initiation time in the message having the *highest* next initiation time of any message in a queue. . .". The reason for using the *highest* next initiation time may not be clear. Because of the ascension numbers, the merge element knows about all messages up to a certain time. The highest initiation time is associated with the last message of that set. That time is the earliest time that the disperse element can initiate the next message. Therefore, the earliest that a message can be received over channel x is $T_{ny} + p_x$. Using this prediction cannot result in a message being processed out of order.

Whenever messages are queued on all of the channels, a message from one of the channels can be chosen. Only when one of the channels becomes empty of messages is it necessary to make a prediction. When one channel is empty, the receipt of a message on the other channel causes the time of the empty channel to increase to at least the value of the next initiation time in that message. A possibility exists that even though the channel time of the empty channel increases, it still does not allow messages from the other channel to be processed. However, one of three things must occur. First, a message can be received on the presently empty channel, in which case a message can be chosen to process. Second, a termination message can be received. The termination message sets the channel time to infinity and causes that time to remain. The time of infinity does not prevent any of the other channels from processing messages. When all channels read infinity, processing stops. Third, another message can be received on the channel already having messages, which allows another prediction to be made, generally raising the time on the empty channel, possibly to the level at which a message can be processed.

The above shows that as long as messages continue to come from some source to P_{f1} up through the termination message, the feedforward network will process to completion and is not the source of a deadlock.

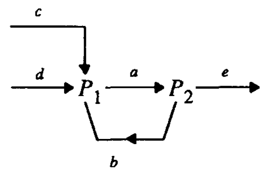
P_{f2} processes messages in order of ascension number along a channel, but not necessarily in order of ascension number when all channels are considered. Ascension numbers normally die at merge points. However, when the feedforward network is embedded in a larger system, a resynchronization may be required. That is, P_{f2} may have to assign new ascension numbers that are in the correct order if another merge can take place beyond P_{f2}. This aspect is discussed later.

V. Simple Feedback Network

A *simple feedback network* from node N_{b1} to node N_{b2} consists of a forward path p_a, from N_{b1} to N_{b2}, a feedback path p_b from N_{b2} to N_{b1}, one or more input paths to N_{b1}, and one output path from N_{b2}.

In a manner similar to the above, let the simulation time at which P_{b1} completes processing of a message be T_n, and let the minimum simulated delay along p_a be t_a and along p_b be t_b. These minimum simulation times are the sums of the minimum simulated delays which are incurred by each process along the path, where t_a includes the minimum simulated delay for P_{b1} and t_b includes the minimum simulated delay for P_{b2}. Assume that P_{b1} obtains the times t_a and t_b during initialization and that P_{b2} obtains the time t_b during initialization. Although it was not important for the feedforward case, consideration must be given to the input and output of the feedback network. Let two inputs to P_{b1} be paths p_c and p_d and let the output from P_{b2} be path p_e.

The picture is as follows:



Let the channel times for channels c and d be 0 initially and let a message appear at time T_c along input channel c and at T_d along channel d. Then a deadlock results immediately, because channel b has a time of 0. However, a prediction can be made as to the earliest that a message can be received along p_b. That time is $T_n + t_a + t_b$, where T_n is allowed to take on the smallest time over all the input channels until the first message is received. The prediction is used to avoid deadlock and allows the message on the channel having the lowest time to be processed. Let a message sent by P_{b1} contain the following information:

1) T_n.

2) A tag which indicates that item 1 is to be passed along until N_{b1} is reached again.

3) Timestamp.

4) The message itself.

As with the feedforward case, each process must send out exactly one message for each input message received, with the exception of P_{b2}. P_{b2} is required to send a NULL message along p_b whenever it would not send a message along that path in response to an input message. Also, P_{b2} is required to send a NULL message along p_e whenever it would not send a message along that path in response to an input message. Thus, P_{b2} always sends out exactly two messages, at least one of which must be null, in response to an input message. The purpose of the NULL messages is to update time on the channel.

The actions of both P_{b1} and P_{b2} need to be specified. The assumption will be made that N_{b1} is not a merge element; therefore, ascension numbers on p_c and p_d are not needed. Channels c and d are considered input channels and channel b is considered a feedback channel in what follows. Finally, t_b is assumed to include the minimum simulated delay for the process P_{b2}.

A Simple Algorithm Defining Actions Taken by P_{b1} for a Simple Feedback Network

1) Set COUNT = 0.
2) If no new messages have been received wait for a message; otherwise, place any messages that have been received in the corresponding queue and go to Step 3.
3) For each channel, determine the message in its queue, if one exists, with the lowest timestamp and assign that timestamp to the channel. If the feedback channel does not have a message in its queue and COUNT = 0, assign the time $T_l + t_a + t_b$ to the feedback channel, where T_l is the local time of the process, unless that time is earlier than the time presently on the feedback channel. Update local time. If the channel having the earliest time has a message in its queue, go to Step 4; otherwise, go to step 2.
4) Process the message on the channel having the earliest time. If a message from an input channel has been processed, increment COUNT by 1 and go to Step 3. If a NULL message from the feedback channel has been processed, indicating P_{b2} had sent a real message over its output channel, decrement COUNT and go to Step 3. If a normal message from the feedback channel has been processed, COUNT remains unchanged.

The purpose of COUNT is to keep track of the number of messages in the feedback loop. COUNT is initialized to 0, since messages can be initiated only by sources external to the feedback loop. When COUNT = 0, meaning no messages are in the loop, the earliest time that a message can be received on the feedback channel is the local time of N_{b1} plus the minimum time around the loop. If this minimum time is used for the feedback channel and the next message is selected according to Steps 3 and 4, messages cannot be processed out of order. Step 4 keeps correct track of the variable COUNT.

An improvement in prediction can be made that allows prediction even when one or more messages are in the loop. Information regarding present initiation time must be retained and be included in the message sent by P_{b1}. In the following, TIMES is an ordered set of present initiation times.

Alternate Algorithm Defining Actions Taken by P_{b1} for Simple Feedback Network

1) Set TIMES = Φ, the null set.
2) If no new messages have been received, wait for a message; otherwise, place any messages that have been received in the corresponding queue and go to Step 3.
3) For each channel, determine the message in its queue, if one exists, with the lowest timestamp and assign that timestamp to the channel. If the channel with the lowest time is the feedback channel and it does not have a message in its queue, assign the time $min(TIMES) + t_a + t_b$ to the feedback channel, where $min(TIMES)$ is the minimum time over set TIMES, unless that time is smaller that the present time on the feedback channel. Update local time. *Note:* If TIMES is empty, then the local time T_l is determined first and substituted for $min(TIMES)$ in the expression. If the channel having the lowest time has a message, go to Step 4; otherwise, go to Step 2.
4) Process the message on the channel having the lowest time and place the present initiation time of the message, T_i, into TIMES. If the message was received over the feedback channel, determine the initiation time, T_i, and remove it from TIMES. Go to Step 3.

The purpose of the set TIMES is to keep more information about messages in the loop to allow for better prediction. The objective is still to determine the earliest time that a message can arrive on the feedback channel. That time is the initiation time of the oldest message in the feedback loop plus the minimum time around the loop. If this predicted time is used, messages cannot be processed out of order. Step 4 keeps track of the correct set of times.

Algorithm Defining Actions Taken by P_{b2} for Simple Feedback Network

1) If no message is present in the queue, wait for a message; otherwise, process the next message in timestamp order.
2) If the output message is to be sent over the output channel only, send a NULL message over the feedback channel to N_{b1} with a timestamp that is the same as that for the message sent over the output channel. (Each process along this path adds its minimum simulated time to its new local time to create the timestamp it sends.) If the output message is to be sent over the feedback channel only, send a NULL message over the output channel with a timestamp that is the same as that for the message sent over the feedback channel. Go to Step 1.

Step 2 ensures that both the normal output and the feedback output receive a message in order to update the channel clocks. If a normal message is not sent, then a NULL message is sent. In this way, P_{b1} can account for all messages introduced into the loop by either receiving a normal message or a NULL message. The NULL message tells P_{b1} that a normal message has exited the loop. Channel time updates are made by having each process along the way add its minimum simulated delay to its new local time.

Intermediate nodes again have only one input. Therefore, if Assumption 2 is satisfied, these nodes process information in correct order. N_{b2} also has only one input; therefore, it processes messages in the correct order. In order to ensure that P_{b1} receives a time update and knows how many messages are in the loop, P_{b2} sends a NULL message back to P_{b1} whenever it sends a regular message over its output channel. The purpose of sending a NULL message over the output channel whenever it sends a message over the feedback channel is to update the output channel time.

The basic concept is still that of choosing the channel with the lowest timestamp. However, where a time is not known for the feedback channel, because a message is not present, a minimum time is predicted if certain conditions are met. That time is the local time plus the minimum simulated delay around the loop. This predicted time can be smaller that the present channel time, in which case the prediction is ignored. In the case of the simple algorithm, the condition is that no messages be in the loop. In the case of the alternate algorithm, the prediction is based on the oldest message that is yet in the loop. To show that P_{b1} processes messages in the correct order, it is only necessary to show that the time prediction on the feedback channel is acceptable.

Consider the simple algorithm. As long as messages continue to arrive on the input channels, messages will continue to progress through the feedback network. When a message is present on each channel, the one with the smallest timestamp is chosen. When a message is not present on the feedback path, there are two possibilities: if COUNT is not 0, one or more messages are in the loop and these will eventually return to the feedback input. A message can be sent when the channel having the smallest time has a message. If COUNT is 0, no message can be forthcoming on the feedback channel. In that case, a prediction is made for that channel, which is the local time plus the minimum time around the loop. The time must be larger than the channel time of the channel having the minimum timestamp. Therefore, the message having the minimum timestamp can enter the loop.

VI. Networks Decomposed into Series, Feedforward, and Feedback Networks

Suppose that a given network is decomposable into series, feedforward, and feedback networks, where one type may exist inside another. We will continue to distinguish between P_1 and P_2 of the feedforward network from those of the feedback network, by using an f subscript to indicate feedforward and a b subscript to indicate feedback. Consider the case of a feedback network within a feedforward network. As long as the feedback network is a simple network, P_{b1} is capable of accepting at least one message and starting it through the network. According to the rules, a message must eventually make a complete loop and return to the feedback input, at which time the channel time can be updated and another message allowed to pass through the network. Thus, a simple feedback network within a feedforward network does not create a problem in terms of deadlock.

Now suppose we have a simple feedforward network inside a feedback network. For the feedback network to continue to function, a message must return for each message that enters the loop. Since the feedforward network can hold messages, a deadlock can result. Therefore, NULL messages may occasionally have to be sent to prevent that deadlock. P_{b1} is able to determine when a deadlock is possible and can then send these NULL messages.

NULL messages must be sent by P_{b1} when it cannot continue because the time on the feedback channel is the lowest time and the feedback channel has no message. Deadlock occurs because, in spite of the predictions that P_{f2} makes on its input channels, it cannot process any of the waiting messages. Therefore, P_{b1} will reach a point at which it can no longer process messages or at least perceives that it cannot because it does not know whether a message is coming on the feedback channel. This situation can be broken by one or more NULL messages from P_{b1}. *Note:* P_{b1} does not send the NULL message unless all its input channels have a message, the feedback channel does not have a message, and the time on the feedback channel is less than that of any input channel.

P_{f1} responds to the NULL messages by attempting to flush out some messages from the feedforward network by sending a NULL message on *each* of its output channels with a next initiation time that is its local time. The timestamp is the local time plus the minimum simulated delay for this process. The reason for sending a message on each channel is that P_{f1} does not generally know which channel of P_{f2} is causing the problem.

The algorithms are simplified and NULL messages are reduced if P_{f2} responds to exactly one of these NULL messages. Then P_{b1} knows that exactly one response will be received to its request. A number of different methods could be used for determining to which NULL message of the set that P_{f2} responds. One approach is to respond to only the first NULL message having a particular next initiation time. The timestamp of the NULL message is the updated local time plus the minimum simulated delay introduced by the process. Each process along the way back to N_{b1} does the same. When P_{b1} receives the NULL message, it processes it in the normal way. The channel may or may not be updated sufficiently to allow another normal message to be generated by P_{b1}. If not, another NULL message must be sent to try to flush out a message. Eventually, the time on the feedback channel must be increased sufficiently to allow a real message to be passed through P_{b1}.

Another approach to determining to which NULL message of the set P_{f2} will respond is to keep processing messages until either all the expected NULL messages have been received or until no more real messages can be processed. In this way, P_{b1} receives real messages to process and the NULL message that it does receive provides a better prediction.

Algorithms for the actions taken by nodes of the feedback and feedforward networks can now be defined. As in the case of the simple feedback network, the present initiation time must be included in the message. Note that t_a in Step 3 is the minimum time taken over all possible paths between N_{b1} and N_{b2}.

Algorithm Defining Actions Taken by P_{b1} for Feedforward Network Inside Feedback Network

1) Set TIMES = Φ, the null set.
2) If no messages have been received, wait for a message; otherwise, place any messages that have been received in the corresponding queue and go to Step 3.
3) For each channel, determine the message in its queue, if one exists, with the lowest timestamp and assign that timestamp to the channel. If the channel with the lowest time is the feedback channel and it does not have a message in its queue, assign the time $min(TIMES) + t_a + t_b$ to p_b, where $min(TIMES)$ is the minimum time over set TIMES, unless that time is smaller than the present time on the feedback channel. Update local time. Note: If TIMES is empty, then the local time T_l is determined first and substituted for $min(TIMES)$ in the expression. If the channel having the lowest time is an input channel and has no message, go to Step 2; otherwise, go to Step 4.
4) If the channel with the lowest time has a message, process the message. Place the initiation time of the message into TIMES unless it was a NULL message on the feedback channel. If the message was received over the feedback channel, determine the initiation time from the message and remove that time from TIMES. Go to Step 3. If the channel with the lowest time is the feedback channel and it has no message, send a NULL message on the output channel having an initiation time that is the local time and a timestamp that is the local time plus the minimum simulated delay of the process. Place the initiation time into TIMES. Go to Step 2.

When the feedback channel does not have a message, channel time is predicted in the same way as for the simple feedback network. Therefore, messages cannot be processed out of order. The NULL message mentioned in Step 4 is necessary because a message can get stuck inside the feedforward loop. Since the possibility exists, an attempt must be made to avoid deadlock by continuing to increase the time on each channel in the loop. The possible deadlock is to increase the time on each channel in the loop. The possible deadlock is detected by the fact that the feedback channel has the lowest time but does not have a message. Therefore, P_{b1} is unable to introduce any more messages into the loop.

Algorithm Defining Actions Taken by P_{f2} for a Feedforward Network Inside a Feedback Network

1) If a message has arrived whose ascension number is *not* the next one in order, wait for the next message. Otherwise, place the message in a queue for that channel. Continue placing received messages in a queue for the corresponding channel as long as the ascension number is the next in sequence. Then go to Step 2.
2) For each channel, determine the message in its queue, if one exists, with the lowest timestamp and assign that timestamp to the channel. If no message exists in the queue for channel x, assign the time $T_{nv} + t_x$ to p_x, where T_{nv} is the next initiation time in the message having the *highest* next initiation time, unless the predicted time is less than that already on the channel. A prediction is similarly made for all the other channels which do not have a message in their queue. Update local time. If the channel with the lowest time does not have a message, go to Step 1; otherwise, go to Step 3.
3) Process the message having the lowest channel time. If the message just processed was a normal message, return to Step 2. If the message processed was the first NULL message with that next initiation time, send a NULL message on the output channel, which is intended for P_{b1}. The timestamp of the NULL message is the local time of the process plus the minimum simulated delay introduced by the process. (Each process along the way back to P_{b1} will do the same). If this NULL message is not the first having this next initiation time, do not send any messages. Go to Step 2.

The algorithm for P_{b2} is the same as the algorithm for the case of a simple feedback network. The algorithm for P_{f1} is the same as that for the simple feedforward case, except that when P_{f1} receives a NULL message from P_{b1}, it sends out a NULL message over *each* channel, as noted earlier.

VII. Feedforward Networks Inside Other Feedforward Networks

Feedforward networks inside other feedforward networks create a special problem, but one which is not difficult to handle. An example of such a case is shown in the following figure.

$$\begin{tikzcd}
& & P_3 \arrow[dr, "f"] & & & \\
P_0 \arrow[r, "a"] \arrow[dr, "b"'] & P_1 \arrow[ur, "c"] \arrow[r, "d"] & P_4 \arrow[r, "g"] & P_6 \arrow[r, "i"] & P_8 \arrow[r, "k"] & {} \\
& P_2 \arrow[r, "e"] & P_5 \arrow[r, "h"] & P_7 \arrow[ur, "j"'] & &
\end{tikzcd}$$

The first problem is that P_6 will never see some messages, those that go over the lower channel. Since it will not see some ascension numbers, it will not be able to proceed, according to the algorithm. The solution is for P_1 to generate its own set of ascension numbers, but retain, in the message, the old ascension numbers to be used later by P_8. Ascension numbers generated by P_1 will die at P_6. However, yet another problem occurs as is explained next.

The order in which messages leave a merge element is not necessarily the same as the order in which the messages were originally sent. That is, messages are handled in timestamp order, not in order of ascension number. A feedforward network generally reorders the messages in time. However, the outer feedforward network is not aware of this reordering in the inner feedforward network. Unless action is taken, P_8 in the above example may receive messages along Channel i with ascension numbers out of order.

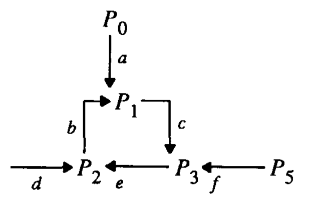
If the messages were to arrive in order of ascension number, there would be no problem. Therefore, consider a message having ascension number 4 arriving before a message having ascension number 2, where message number 4 has a lower timestamp. The algorithm requires that P_8 wait for message number 2 to arrive before it can place message numbers 2, 3, and 4 into their respective queues. That means that channel times cannot be predicted at P_8 based on these messages until after message 2 arrives. However, a view of the total picture indicates that because message 2 has a later timestamp, P_8 need not wait until message 2 arrives to make a prediction on channels not having a message. Note that the only adverse effect is that P_8 must wait longer to process messages. It does not erroneously process messages out of order.

One solution is to have P_6, in the above example, reorder the message numbers according to the order in which it processes the messages. Assume P_0 assigns global numbers and P_1 assigns local numbers. Some notation will be helpful. Let (l, g) be a message designation in which l is the local ascension number and g is the global ascension number. Suppose P_6 receives the message (2, 5). It cannot proceed until it receives the message $(1, x)$. Let that message be message (1, 3). Finally assume that the message (2, 5) has the lower timestamp. Then the reordering of messages caused by the merge element P_6 should also cause a reordering of the global message numbers, so that the message designated (2, 5) is sent with global ascension number 3 and the message designated (1, 3) is sent with global ascension number 5. Then the merge element P_8 can proceed sooner than it could otherwise. All global message numbers passing through the local merge element, in this case P_6, are preserved, but the local merge element puts ascension numbers back into correspondence with the timestamps.

The data can be obtained by an inspection of the messages waiting in the queues for the various channels. If each message up to and including some local ascension number has been placed into one of the queues, then all the messages that had been sent up to the time of the last message are known. The global ascension number of each of these messages can be determined and placed in a list. As messages are processed, the ascension numbers can be taken in proper order from the list. This reordering need not be perfect, since correct operation will still occur, but processing can continue sooner if the ascension numbers are reordered in the best way.

VIII. Treatment of Loops with Sources

A more general structure to consider is the loop with sources such as the one in the following figure.



Suppose all channels are initially 0. Then as messages arrive from P_0, P_4, and P_5, a deadlock is likely to occur, since all the feedback channels read 0 and the other channels are likely to read something greater than 0. In this case, no process can start processing. However, each process is capable of making an estimate on the earliest that a message can arrive over the channel in the loop. Consider P_2. If P_2 knows the minimum simulated delay of P_3, it knows initially that it cannot receive a message from P_3 until that minimum delay has elapsed. Let t_i be the minimum simulated delay of process P_i, where the sources are assumed to have 0 simulated delay. Thus, P_2 can set the time of channel e to t_3. P_1 and P_3 can similarly set channels b and f to t_2 and t_1, respectively. Making these predictions may or may not allow messages to be processed. Each of the three processes can compare the loop channel times to the source channel times to determine if it can process a message. If it cannot, the only choice is to send a NULL message to attempt to update its loop channel time. The timestamp of the NULL message is determined by taking the local time and adding the minimum simulated delay to it.

The channel, once it has the first message, can continue to estimate the time on the loop channel when a message is not present on that channel. It does so by adding the known minimum simulated delay for the process ahead of it in the loop to the timestamp of the last message it received over the channel, provided that timestamp was determined from a real message, and *not* a NULL message. The reason it must be a real message is that a real message presents the actual time the message is received. Thus another message cannot arrive until the minimum simulated delay has passed. However, with a NULL message, the time is an estimate of the earliest time at which the next message can arrive. The next message could, in fact, arrive at that estimated time or shortly thereafter. Thus the minimum simulated delay cannot be added in this case. As with all other cases of estimates, the estimates can be low, so that if the present channel time is later than this estimate, the present channel time is kept. *Note:* Each of the processes in the loop should wait for a message from the associated source before proceeding.

The above approach does not completely avoid the need for NULL messages, but it does reduce the number that are needed. If the simulated delays are small, so that a large number of traversals of the loop would have to be made before a message could be processed from one of the sources, then some other approach might be in order.

IX. Alternate Models for Queues

The FIFO (or FCFS) queue is a natural one to use in Misra's method for choosing messages to process. However, the above framework allows other types of queue (scheduling disciplines) to be used in place of the FIFO. Examples are Last In, First Out (LIFO or stack), Priority queue, and Shortest Remaining Processing Time (SRPT). LIFO should be self-explanatory. In a priority queue, messages are selected based on some priority, which could be made part of the messages. For example, certain messages could be considered urgent and have a high priority. The SPRT discipline selects messages corresponding to the process that is determined to take the shortest time. For example, if the queue is a printer queue, the job that is determined to be the one requiring the least amount of time to print would be the one selected first. The determination may be based on an approximation, such as the number of lines or the number of bytes in the file.

A problem, which is an extension of the problem associated with FIFO's, is encountered with the merge element of a feedforward network which implements a type of scheduling discipline other than FIFO. All messages up to a certain time must be present before proceeding. In general, with any scheduling discipline other FIFO, all messages up to and including the local time of the process must be present before selecting a message to process.

In order to obtain some understanding of the problem, consider a scheduling discipline based on priorities. Suppose that messages are of two priorities, 1 and 2, with 1 being the higher priority. Let channel 1 receive priority 1 (type 1) messages and channel 2 receive priority 2 (type 2) messages. A selection algorithm for choosing a message to process can be developed for this scheme. Each channel has a time associated with it and the process (including the queue) has a local time associated with it, where the local time can be greater than any of the channel times. Let channel 1 have time t_1, channel 2 have time t_2, and the process have local time t_l, where $t_2 < t_1 < t_l$. The message from channel 2 would be processed first in a FIFO system. However, since priorities are given to messages, the message having the higher priority should be processed first. Time is still important, since the lower priority message should be processed if a higher priority message is not waiting to be processed.

The existence of a higher priority message can be determined based on local time. In the above example, a type 1 message has arrived after the type 2 message, but before the present local time of the process; therefore, the type 1 message should be processed first. If the order had been $t_1 < t_2 < t_l$, then the type 1 message should be processed first. However, if the order had been $t_2 < t_l < t_1$, the type 1 message has arrived after the present local time, and the type 2 message should be processed first; it is the only one waiting in the queue at the local time of the process. The tacit assumption has been made that the present task, once started, will not be interrupted.

Where NULL messages become necessary, for example, where a feedforward network is inside a feedback network, the NULL message must be assigned a priority. If the highest priority were used, the NULL message(s) would be processed first, causing time to be updated quickly around the loop. If the lowest priority were used, real messages would be taken first, which would slow the updating of time around the loop, but would speed up the processing of more meaningful information within the loop. NULL messages may be given any priority value. However, using a higher or lower value of priority for NULL messages will affect efficiency. Some experimentation with various values of priority for NULL messages would help to determine what priority should be used under what conditions.

Now consider the Last In, First Out (LIFO) queue, also called a stack. The process must know that it has received all messages up to its present time before it can decided which one is the last one received. That means waiting until all such messages are present before choosing a message to process. The message that arrived most recently in the past is the one processed.

Where the feedforward network is inside a feedback network, NULL messages are generally required. The order in which the NULL messages are processed is not important. However, the most reasonable order is the same as that for the scheduling algorithm. A NULL message is processed as before, that is, the local time is updated before processing the NULL message, but not after processing the NULL message, because no real message was processed to cause the local time to be increased by the simulated delay of the process.

In the LIFO case, an old message could remain around for a long time before being processed. Node P_{f1} of the feedback network can process messages up to the point where it must receive a message along the feedback path. As with the FIFO case, a point can be reached at which deadlock occurs. Potential deadlock can be detected by P_{b1} of the feedback network. If the time on the feedback channel is the smallest time and it does not have a message, then a NULL message must be sent. Note that if an old message remains in the system without being processed, the algorithm for predicting the earliest time a message can be received on the feedback channel will not be effective.

Yet another type of scheduling discipline is the Shortest Remaining Processing Time (SRPT). In this case, priority is based on the operation to be performed or on the data itself. A choice of the next message to process still requires that all messages up to and including the local time be available. In that sense, it is not significantly different in the way it affects the simulation than the others.

Predictions should be used, as before, to determine the earliest time at which a message could arrive. If a message of a higher priority could arrive before the present local time, then the process must wait until enough messages have been received to ascertain whether such a message does or does not arrive.

X. Summary

A method has been presented for reducing the number of NULL messages which must be sent in Misra's method of Distributed Discrete Event Simulation. The specific structures that were investigated are feedforward and feedback networks and combinations of one inside the other. A loop structure was also considered. Prediction of channel times and ascension numbers were used to avoid sending some NULL messages. The type of merge element normally used is the FIFO queue. However, it was shown that other types of queue can be used instead.

A problem yet to be solved and one that is presently being investigated is the problem of taking a process flow graph and decomposing it into structures of the form discussed in the paper. With the capability of decomposing the process flow graph into its constituent components, an automated procedure could be developed that would automate the development of algorithms that predict channel values, create and destroy ascension numbers, and in general control the flow and synchronization of messages within the system for each process.

References

[1] D. R. Jefferson, "Virtual time," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 3, pp. 404-425, July 1985.

[2] K. M. Chandy and J. Misra, "A case study in design and verification of distributed programs," *IEEE Trans. Software Eng.*, vol. SE-5, no. 5, pp. 440-452, Sept. 1979.

[3] J. Misra, "Distributed discrete event simulation," *ACM Comput. Surveys*, vol. 18, no. 1, pp. 39-65, Mar. 1986.

[4] J. Peacock, J. W. Wong, and E. G. Manning, "Distributed simulation using a network of processors," *Comput. Networks*, vol. 3, no. 1, pp. 44-56, Feb. 1979.

[5] B. Jayaraman, "Dataflow approach to discrete simulation," in *Proc. 1981 Parallel Processing Conf.*, IEEE, 1981, pp. 158-159.

[6] D. Kumar, "Simulating feedforward systems using a network of processors," in *Proc. Nineteenth Annu. Simulation Symp.*, IEEE Comput. Soc., Tampa, FL, Mar. 12-14, 1986, pp. 127-144.

Ronald C. De Vries (S'67-M'67-SM'87) was born in Chicago IL, on December 4, 1936. He received the B.S. degree from Northwestern University, Evanston, IL, and the M.S. and Ph.D. degrees from the University of Arizona, Tucson, all in electrical engineering, in 1959, 1962, and 1968, respectively.

From 1959 to 1960, he worked for the Datastor Division of Cook Electric, Skokie, IL. He was a member of the Electrical Engineering Faculty at San Diego State College, San Diego, CA, from 1964 to 1966, and has been a faculty member of the Department of Electrical and Computer Engineering at the University of New Mexico since 1967. During the summer of 1968, he performed research at NASA's Electronic Research Center on a NASA-ASEE summer faculty fellowship. His research interests include fault detection and diagnosis, fault tolerant computing, and system safety.

Dr. De Vries is a consultant for Sandia National Laboratories and is a member of Sigma Xi, Eta Kappa Nu, and the IEEE Computer Society.