THE YADDES DISTRIBUTED DISCRETE EVENT SIMULATION SPECIFICATION LANGUAGE AND EXECUTION ENVIRONMENTS

Bruno R. Preiss

Department of Electrical Engineering
University of Waterloo,
Waterloo, Ontario, Canada, N2L 3G1

ABSTRACT

This paper describes a simulation specification language and execution environments that are being used to study the performance of distributed discrete event simulation. First, a simulation programming model based on Chandy-Misra distributed discrete event simulation is presented. Then, the Yaddes simulation specification language is described. Yaddes specifications are translated into C language programs which are then compiled and linked with a run-time support library. Next, the implementation of the run-time support libraries is described. The four libraries currently supported are: (1) traditional, event-list-driven discrete event simulation, (2) distributed discrete event simulation based on multiple, synchronized event lists, (3) Chandy-Misra distributed discrete event simulation, and (4) virtual-time-based distributed discrete event simulation.

1. INTRODUCTION

The Yaddes† system is a tool for constructing discrete event simulations. The principle features of the Yaddes system are:

1. the Yaddes simulation specification language and compiler,
2. run-time libraries that support various simulation execution mechanisms.

The Yaddes user prepares a specification of the desired simulation. Yaddes then translates the specification into a C language (Kernighan and Ritchie 1978) program. This program is then compiled and linked to an execution mechanism library to form a complete program that performs the desired simulation.

The advantage of the Yaddes system is that it uses a programming model that supports several different simulation execution mechanisms. In particular, the mechanisms currently provided are:

1. traditional (event-list-driven) discrete event simulation,

† Yet Another Distributed Discrete Event Simulator
2. distributed discrete event simulation based on multiple, synchronized event lists,
3. Chandy-Misra distributed discrete event simulation (Misra 1986), and
4. virtual-time-based (Jefferson 1985) distributed discrete event simulation using the time warp (Jefferson et al. 1987) mechanism.

The Yaddes user need not be concerned with the execution mechanism used. A Yaddes specification can be executed using any of the mechanisms merely by linking to the appropriate run-time library. Provided that the specifications are coded properly, the results of the simulation are independent of the execution mechanism used.

This paper is an overview of the implementation of the Yaddes system. A programming model that supports the various execution mechanisms is described in Section 2. The Yaddes programming language is very briefly described in Section 3. (The Appendix contains a complete Yaddes program that illustrates the use of many of its features.) The implementations of the four execution environments listed above are described in Section 5.

2. THE YADDES PROGRAMMING MODEL

The programming model used in the Yaddes system is based on Chandy-Misra distributed discrete event simulation. The real-world system is modeled by a collection of physical processes that periodically exchange information. This exchange of information occurs at discrete points in time. Every instant at which one process provides information to another is called an event.

The physical processes of the real-world system are mimicked in the simulation by computer programs called logical processes. The exchange of information by the physical processes is mimicked in the simulation by the exchange of messages by the logical processes. Since the computer simulation does not execute in real time, each logical process has its own notion of time and each message is time-stamped with the time of the corresponding real-world event.

The elements of a Yaddes logical process are shown in Figure 1. A logical process in the Yaddes system is an instance of a model. The use of the model abstraction allows the specification of generic logical processes. For example, a model might be used to describe a queue and server. The simulation of a queueing system consisting of several queues and servers could then be constructed by using multiple instantiations of the model.

A model simply describes a general state machine. A general state machine has an arbitrary (finite) number of inputs and outputs and a (finite, albeit possibly large) set of states. For example, the model shown in Figure 1 has three inputs and three outputs.

In the Yaddes system, event messages have the following format:

| time | value |

The time field contains the time stamp of the event. The semantics of the value field are not specified by Yaddes. The simulation programmer is free to assign any meaning to the value field. (The value field is declared as a C language “int” type.)

The state machine is driven by the occurrence of event combinations. An event combination is a collection of one or more input events having the same time stamp. In response to an event combination, a model may change its state and produce zero or more output events on each
of its outputs. The Yaddes system ensures that event combinations are presented to the model in increasing time-stamp order. (N.B., all events with the same time stamp are presented to the model in a single event combination.) Output events generated by the model may be time-stamped with any time not less than the time stamp of the input combination. However, output events on each output arc must also be produced in increasing time-stamp order.

In the Yaddes system, the connections between logical processes are static. Each input of every logical process must be connected to the output of some other logical process. That is, logical processes require unity fan-in. The output of a logical process may be connected to zero or more logical processes. That is, logical processes may have arbitrary fan-out. In the Yaddes system, the connections between processes are represented by output tables as shown in Figure 1. Each output of every logical process has an output table which specifies the logical processes to which that output is connected. Each entry in an output table has the following format:

| processor ID | process ID | input number |

The first two fields uniquely identify a logical process. The third field specifies an input to that logical process. (Note that, in a multiprocessor Yaddes environment, logical processes are statically assigned to processors and do not migrate during the course of the simulation.)

3. THE YADDES LANGUAGE

The Yaddes language is a specification language in the style of Yacc (Johnson 1979) and Lex (Lesk 1979). That is, it is a specification language based on the C programming language. Yaddes source files are translated into C language programs that are then compiled and linked to a Yaddes run-time support library to produce an executable simulation program.

The basic components of a Yaddes program are model specifications, process specifications, and connection specifications. Model specifications are used to describe general state machines. Process specifications are used to create logical processes by instantiating models. Connection specifications are used to describe the connections between the inputs and the
outputs of the logical processes.

**Model Specifications.** A model is completely specified by (1) enumerating its inputs and outputs, (2) describing the state and giving an initial state, and (3) giving a specification of the actions of the general state machine. Yaddes translates the state and initial state specifications into a C structure type declaration (“typedef struct”) and an initial value for that structure.

The actions of the general state machine are specified by enumerating the event combinations of interest and by associating with each event combination a sequence of C language statements that are to be executed whenever the event combination occurs. In addition to event combinations, initial, final, and default actions can be specified. The initial and final actions are executed before the simulation begins and after the simulation ends (respectively). The default action is invoked whenever an event combination occurs for which an action has not been explicitly declared. The statement sequence can examine and alter the state, and generate output events using Yaddes library routines. Yaddes translates the action specifications into a (reentrant) C language procedure. This procedure is invoked by the run-time environment on every input event combination.

**Process Specifications.** Logical processes are instantiations of models. A logical process is completely specified by (1) giving it a name, (2) specifying the model of which it is an instance, and (3) specifying the number of the processor to which the logical process is to be assigned. Yaddes translates the process specification into the declarations of a state variable and output table arrays. The type of the state variable is that given in the model state specification and the initial value of that variable is that given in the model initial state specification.

**Connection Specifications.** Each connection specification is simply an enumeration of the output table for an output of a logical process. Yaddes uses the connection specifications to initialize the output table arrays.

### 4. EXECUTION ENVIRONMENTS

As stated earlier, the Yaddes programming model requires that the input event combinations be presented to the model in increasing time-stamp order and that all simultaneous events be grouped together into a single event combination. However, since the time stamps of output events are arbitrary and since connections between logical processes are arbitrary, some sort of synchronization is required to ensure that the inputs are presented in the correct order. This is the purpose of the execution environment.

The Yaddes system currently supports the four different execution environments listed in Section 1. The Yaddes system hides the details of the underlying simulation execution mechanism so that the user need only be concerned with the specification, not the implementation of the simulation. There are two important advantages of the ability to support different execution mechanisms:

1. By executing the same specification using different execution mechanisms, it is possible to directly and quantitatively compare the performance of the execution mechanisms (Preiss et al. 1988).

2. The user can change the execution mechanism used without having to recode the simulation specifications. In this way, the most efficient mechanism can be chosen empirically.
The various execution environments are implemented as run-time support libraries. Thus, a different execution environment can be used simply by linking the compiled simulation code to the appropriate library. The implementations of the four execution environments are described in the following sections:

4.1. Event-List-Driven Simulation

The event list driven simulation environment uses the traditional discrete event simulation mechanism. A single data structure, called the event list is used to hold future events (see Figure 2). The event list is sorted by time and by logical process ID. The basic execution cycle involves removing events from the head of the event list, forming event combinations, and causing the appropriate logical processes to perform the action associated with the given input event combination. When an action causes output events, those events are inserted into the event list.

The current implementation of the event-list-driven mechanism uses a linear list to hold the future events. It is well known that this is not the optimal data structure for this application. In order to obtain uninflated speedup figures, a more efficient data structure such as calendar queues (Brown 1988) should be used.
4.2. Multiple, Synchronized Event Lists

The multiple, synchronized event list execution environment is a simple extension of the basic event list mechanism on a multiprocessor. The basic event list execution mechanism is essentially a serial algorithm in that event combinations are processed one at a time. However, it is not strictly required that the event combinations be executed one at a time. In particular, all the event combinations having the same time value as the first event in the list could be executed in parallel. The multiple, synchronized event list environment attempts to exploit the parallelism at the head of the single event list by partitioning the event list. In this mechanism, each processor has its own local event list. In addition, one processor has special status and acts as a global synchronizer (see Figure 3).

![Figure 3. The Multiple, Synchronized Event List Environment](image)

The basic execution cycle consists of the following steps:

1. Each processor sends a message to the synchronizer containing the time-stamp of the next event on the local event list.

2. The synchronizer selects the minimum next event time and broadcasts this time.

3. Each processor having this minimum value removes events from its event list, forms event combinations, and invokes the appropriate logical processes’ actions. Actions causing output events either cause that event to be inserted into the local event list or cause a message to be sent to another processor requesting that it insert an event into its local event list.
4. When a processor is finished executing all the actions for a given value of simulation time, it sends a completion message to all its successors indicating that it is done.

5. Finally, each processor waits until it receives a completion message from all its predecessors before it goes to back to step 1.

The last two steps of the basic cycle are crucial for the correct operation of this execution environment. These steps ensure that all events have been posted on local event lists before returning to step 1. There are a number of ways to ensure that all events have been posted on local lists before returning to step 1. These include:

1. When each processor finishes step 3, it sends a completion message to the synchronizer. The synchronizer waits until it has received a completion message from all the processors and then sends a completion message to each processor. When a processor receives a completion message, it proceeds to step 1. In order for this scheme to operate correctly, messages sent from a processor must be delivered in exactly the same order in which they are sent regardless of their destination. This requirement is difficult to achieve in practice.

   This scheme requires $2N$ messages in every cycle for synchronization, where $N$ is the number of processors.

2. When each processor finishes step 3, it broadcasts a message to all processors indicating that it is done. In order for this scheme to operate correctly, messages between pairs of logical processes must be delivered in the order in which they are sent.

   This scheme requires $N(N-1)$ messages in every cycle for synchronization in a point-to-point message passing system, where $N$ is the number of processors. Note that the cost of doing a broadcast in a point-to-point message passing system becomes prohibitive when the number of processors becomes large.

3. Each processor sends a completion message to each of its successors. (Its successors are those processors to which the outputs of local logical processes are connected.) It then waits until it receives a completion message from each of its predecessors. (Its predecessors are those processors to which the inputs of its local logical processes are connected.) This is the method used in Yaddes implementation.

   A processor can determine the identities of its successors simply by examining the output tables of its logical processes. Furthermore, each processor need only count the number of completion messages it receives from its predecessors in each cycle. Since a processor cannot determine how many predecessors it has from its own output tables, the Yaddes translator pre-computes the count.

   This scheme requires $N(N-1)$ messages in every cycle for synchronization in the worst case.

4.3. Chandy-Misra Distributed Discrete Event Simulation

In the Chandy-Misra distributed discrete event simulation environment each logical process runs as a separate task on a separate processor. In this environment, the logical processes consist of the Yaddes model, state, and output tables, and an envelope (see Figure 4).

The basic execution cycle begins when an envelope receives a message. The envelope buffers messages until an event combination can be formed. The decision regarding when an event combination can be formed is made purely on information available to the envelope.
Specifically, an event combination with simulation time $t$ can only be formed when the envelope has received an event message for each input of the associated model having time $t'$, such that $t' \geq t$. When an event combination is formed, the appropriate action is invoked. When an action causes an output event, a message is sent to the envelope of the appropriate logical process.

This execution environment has the potential for deadlock. That is, simulations that do not deadlock under the other execution mechanisms, may deadlock under this mechanism. Although it is possible to implement deadlock detection and recovery mechanisms, the current implementation of Yaddes requires that the simulation programmer explicitly code the simulation specifications in a way that prevents deadlock (if this environment is to be used).

The Yaddes system provides two approaches to the prevention of deadlock:

1. **Null messages.** Null messages are superfluous events inserted by the programmer. The meaning of such messages is “no event has occurred at time $t'$.” This allows an envelope to determine that it has received all the events up to a given time $t'$ on a given model input.

2. **Explicit control of the envelope.** In some cases, it is possible for the model itself to know that no event will occur on one or more of its inputs. In such cases, the envelope can be informed that no event will occur, and the envelope may safely ignore that input.

Both of the preceding mechanisms are supported by the Yaddes system. Note that in this case, the simulation programmer must be aware of the underlying execution mechanism and explicitly prevent deadlock. This can be accomplished by calling special library routines from the model’s actions. The routines provided to assist in the prevention of deadlock have no effect in the other execution environments. Hence, a simulation written for the Chandy-Misra environment can be executed under the other execution environments without change.
4.4. Virtual-Time-Based Distributed Discrete Event Simulation

The virtual-time distributed discrete event simulation environment is based on the time warp operating system (Jefferson et al. 1987). As in the Chandy-Misra environment, each logical process runs as a separate task on a separate processor and logical processes consist of the Yaddes model, state, and output tables together with an envelope. However, the virtual-time environment differs from Chandy-Misra in that it uses an event synchronization mechanisms that does not deadlock.

Every envelope has a local clock. When an event message is received by an envelope there are three possibilities — its time stamp is either before, after, or equal to the local value of simulation time. If its time stamp is after the local time, an input event combination (consisting of the single event) is formed and the appropriate action is invoked. However, if the time stamp of the received event message is less than or equal to the local clock value, the envelope has already processed an event combination with time greater than or equal to the incoming event. The envelope must back up to the time of the incoming message. This backing-up is facilitated by an elaborate checkpointing mechanism that allows earlier states to be recovered. Essentially, an earlier state is restored, input event combinations are rescheduled, and output events are cancelled by sending antimessages. The envelope has buffers that save past inputs, past states, and antimessages (see Figure 5).

An important component of the implementation of the virtual-time environment is the fossil collection algorithm. Fossils (Jefferson 1985) are past inputs, states, and antimessages that are no longer required and may be discarded.

The Yaddes fossil collection algorithm uses a circulating token message. Each logical process $P_0, \cdots, P_{N-1}$ maintains two local variables: $LT$, the “current” local time (i.e., the time of the most recent event combination), and $LT_{\text{min}}$, the minimum value of “current” time since the last token message. The token message contains three data: $i$, the ID of the logical process that

![Figure 5. The Virtual-Time Environment](image-url)
is the ‘‘owner’’ of the token, $T_f$, the fossil collection time, and $T_{\text{min}}$, a value used in the calculation of a new value for $T_f$.

Each logical process executes the following algorithm when it receives the token message:

receive $(i, T_f, T_{\text{min}})$ from $P_{(j+N-1) \mod N}$
discard all information older than $T_f$
if $i=j$ then
    $T_f \leftarrow LT_{\text{min}}$
    $T_{\text{min}} \leftarrow LT$
else
    if $T_{\text{min}} < LT_{\text{min}}$
        $T_{\text{min}} \leftarrow LT_{\text{min}}$
        $i \leftarrow j$
    end if
end if
$LT_{\text{min}} \leftarrow LT$
send $(i, T_f, T_{\text{min}})$ to $P_{(j+1) \mod N}$

The token serves two purposes in this algorithm. First, it initiates fossil collection. Whenever a logical process receives the token, it discards all information older than $T_f$. Second, the token is used to compute a new value of $T_f$. The algorithm operates as follows: One logical process is said to ‘‘own’’ the token. This process sets $T_{\text{min}}$ to local time $LT$ and then passes the token on to the next logical process. When a logical process receives the token, it compares $T_{\text{min}}$ with the minimum local time since the last token $LT_{\text{min}}$. If $LT_{\text{min}} < T_{\text{min}}$, the logical process claims ownership of the token and sets $T_{\text{min}}$ to $LT_{\text{min}}$. When a token returns to its owner, it is guaranteed that all states prior to $T_{\text{min}}$ can be discarded. Thus, the owner sets $T_f$ to $T_{\text{min}}$ and the procedure begins anew. (In order for this algorithm to operate correctly, messages between logical processes must be delivered in the order in which they are sent.)

5. SUMMARY

This paper presented a simulation programming model based on Chandy-Misra distributed discrete event simulation. According to this programming model, systems are specified as networks of logical processes which are instances of generic, general state machines. A programming language that is tailored for the specification of such systems was presented. Simulation programs specified in the Yaddes language are translated into C language programs which are then compiled and linked with a run-time support library. The implementation of four run-time support libraries has also been described. The ultimate goal of this work is to study the performance of various distributed discrete event simulation environments. The Yaddes system provides a tool for constructing simulations that can be executed under various execution mechanisms.
6. PROJECT STATUS

Two uniprocessor versions and a multiprocessor version of the run-time libraries are currently implemented:

1. A version of the libraries that runs under BSD4.3 Unix on a DEC microVAX II and simulates a multiprocessor by multitasking under a single Unix process. This version is not portable since it uses machine-specific assembly language routines to implement ‘‘light-weight’’ processes and fast context switches.

2. A portable version of the libraries written entirely in C. This version has been run under BSD4.3 Unix on a DEC microVAX II, under DOMAIN/IX on an Apollo DN3010, and under MS-DOS on a Toshiba 3100 (PC-compatible).

3. The multiprocessor version of the libraries is implemented using Apollo’s Network Computing System (Apollo 1987) to provide a distributed processing environment. This version runs under DOMAIN/IX on a ring of Apollo DN3010 workstations.

7. ACKNOWLEDGEMENTS

This research was funded by the Information Technology Research Center (ITRC) of the Province of Ontario, Canada and by the Natural Sciences and Engineering Research Council (NSERC) of Canada grant OGP0036635.

I would like to acknowledge the input of Wayne M. Loucks, whose input as the first serious user of Yaddes significantly contributed to its evolution.

8. REFERENCES


Jefferson, D. R. 1985. ‘‘Virtual Time.’’ *ACM Transactions on Programming Languages and Systems* 7, no. 3 (Jul.): 404-425.


9. APPENDIX — LATCH CIRCUIT

An example to illustrate the format of a Yaddes specification is given below. The system simulated in this example is a logic network consisting of two NAND gates that implement a latch as shown in Figure 6.

```
#include <stdio.h>
#define GATE_DELAY 10
#define NAND(X,Y) (~(X) & (Y))

%%
model TwoInputNand
  inputs in0, in1
  outputs out
  state { int input0; int input1; int output; }
  initial state { 0, 0, 1 }
  action initial {
    OUTPUT ($out, $time + GATE_DELAY, $state->output);
  }
  action in0 {
    $state->input0 = $event [$in0];
    goto action_in0_in1;
  }
  action in1 {
    $state->input1 = $event [$in1];
    goto action_in0_in1;
  }
  action in0, in1 {
    int tmp;
    $state->input0 = $event [$in0];
    $state->input1 = $event [$in1];
  }
```

![Figure 6. Latch Circuit](image-url)
action_in0_in1:
    tmp = NAND($state->input0,$state->input1);
    if (tmp != $state->output) {
        $state->output = tmp;
        OUTPUT ($out, $time + GATE_DELAY, tmp);
    }
    else {
        NULL_OUTPUT ($out, $time + GATE_DELAY, tmp);
    }
}
end model

model ReadFromFile
    inputs none
    outputs out
    state { FILE *ifp; }
    initial state { NULL }
    action initial {
        int time, event;
        if (($state->ifp = fopen ($name, "r"))
            == NULL) {
            (void) fprintf (stderr, "can’t open %s
", $name);
            exit (1);
        }
        while (fscanf ($state->ifp, "%d%d",
            &time, &event) == 2) {
            OUTPUT ($out, time, event);
        }
    }
end model

model WriteToFile
    inputs in
    outputs none
    state { FILE *ofp; int last; }
    initial state { NULL, -1 }
    action initial {
        if (($state->ofp = fopen ($name, "w"))
            == NULL) {
            (void) fprintf (stderr, "can’t open %s
", $name);
            exit (1);
        }
    }
end model
}  
action in {
    if ($event [$in] != $state->last) {
        $state->last = $event [$in];
        (void) fprintf ($state->ofp,
            "%d %d\n", $time, $event[$in]);
    }
}
end model

process X on 0 : ReadFromFile
process Y on 0 : ReadFromFile
process Gate0 on 0 : TwoInputNand
process Gate1 on 0 : TwoInputNand
process Z on 0 : WriteToFile

connect X.out to Gate0.in0
connect Y.out to Gate1.in1
connect Gate0.out to Gate1.in0, Z.in
connect Gate1.out to Gate0.in1