Yaddes: Yet Another Distributed Discrete Event Simulator

Bruno R. Preiss
Department of Electrical Engineering
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1

Table of Contents
1. Introduction
2. Computer Simulation and Modelling
3. Execution Mechanisms
   3.1. Event List Driven Simulation
   3.2. Multiple, Synchronized Event Lists
   3.3. Chandy-Misra Distributed Discrete Event Simulation
   3.4. Virtual Time based Distributed Discrete Event Simulation
4. Example: Exclusive-OR Circuit
5. The Yaddes Specification Language
   5.1. Model Specifications
      5.1.1. Inputs
      5.1.2. Outputs
      5.1.3. State
      5.1.4. Initial State
      5.1.5. Actions
         5.1.5.1. General Event Combination Actions
         5.1.5.2. Initial Action
         5.1.5.3. Final Action
         5.1.5.4. No Event Action
         5.1.5.5. Default Action
   5.2. Process Specifications
   5.3. Connection Specifications
   5.4. Procedure Specifications
      5.4.1. Initial procedure
      5.4.2. Final procedure
   5.5. Predefined Variables and Constants
      5.5.1. Constants
      5.5.2. Variables
         5.5.2.1. $event
         5.5.2.2. $name
         5.5.2.3. $state
         5.5.2.4. $time
   5.6. Predefined Functions
      5.6.1. ACTIVATE
      5.6.2. DEACTIVATE
      5.6.3. EVENT_OCCURRED
      5.6.4. IGNORE
5.6.5. OUTPUT
5.6.6. NULL_OUTPUT
5.6.7. SET_TIME_LIMIT

5.7. Compiling and Running the Simulation
5.7.1. Compiling a Yaddes Language Specification
5.7.1.1. Separate Compilation
5.7.1.2. Run-time Execution Support Mechanism Libraries
5.7.2. Running a Simulation Program
5.7.2.1. +a
5.7.2.2. +h and -h
5.7.2.3. +i
5.7.2.4. +k
5.7.2.5. +m
5.7.2.6. +o
5.7.2.7. +p
5.7.2.8. +s
5.7.2.9. +t
5.7.2.10. -w

6. Pseudo-Random Number Generator Package
6.1. Using the Pseudo-Random Number Generator Package

7. Statistics Gathering and Reporting Package
8. Avoiding deadlock under the Chandy-Misra execution mechanism
8.1. Using OUTPUT and NULL_OUTPUT
8.2. Using IGNORE
8.3. Using ACTIVATE and DEACTIVATE

9. Example: Queue and Server
9.1. The Arrival Process
9.2. The Queue Process
9.3. The Server Process
9.4. The complete specification

Appendix 1. Yaddes BNF
Appendix 2. Kernel Statistics from event-list.a
Appendix 3. Kernel Statistics from multi-list[vax].a
Appendix 4. Kernel Statistics from chandy-misra[vax].a
Appendix 5. Kernel Statistics from virtual-time[vax].a
Appendix 6. Yaddes Compiler Output

1. Introduction

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

- the Yaddes simulation specification language and compiler,
- run-time libraries that support various simulation execution mechanisms with extensive built-in trace and debug support,
- a pseudo-random number generator package that supports multiple, independent pseudo-random number streams, and
- a statistics collection and reporting package.
The Yaddes user prepares a specification of the desired simulation. Yaddes then compiles the specification into a collection of C language subroutines. These subroutines are then compiled using the C compiler and then linked to an execution mechanism library to form a complete program that performs the desired simulation.

The advantage of the Yaddes system over other discrete event simulation packages is that it uses a unified modelling methodology that supports several different simulation execution mechanisms. In particular, the mechanisms currently provided are:

- traditional (event-list driven) discrete event simulation,
- distributed discrete event simulation based on multiple, synchronized event lists,
- Chandy-Misra distributed discrete event simulation, and
- virtual-time-based distributed discrete event simulation using the time warp mechanism.

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

2. Computer Simulation and Modelling

The purpose of computer simulation is to gain insight into the behaviour of an existing or imagined real-world system. Simulation is used both when the real-world system is too complex for mathematical analysis and to validate mathematical analyses when they are possible.

The modelling methodology used in the Yaddes system is based on Chandy-Misra distributed discrete event simulation. According to this methodology, the real-world system is modelled by a collection of physical processes that periodically exchange information. This exchange of information takes place at discrete points in time. Every instant at which one process provides information to another is called an event.

The computer simulation of the real-world system is obtained by constructing a computer program in which the behaviour of each physical process is mimicked by a computer program. These programs are 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 tagged with the time of the corresponding real-world event.

The Yaddes system provides a language and compiler that aids in the specification of the behaviour of logical processes. A logical process in the Yaddes system is 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. 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 logical process may change its state and produce zero or more output events on each of its outputs. The Yaddes specification language is used to specify the state of a logical process and to associate programs with event combinations. (The programs are written as sequences of C language statements.)

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. The Yaddes specification language provides a means for enumerating the connections between logical processes.
3. Execution Mechanisms

The Yaddes system currently supports the use of four different discrete event simulation mechanisms. (The mechanisms currently supported are described in the following sections.) However, 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. First, by executing the same specification using different execution mechanisms, it is possible to directly and quantitatively compare the performance of the execution mechanisms. Second, 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 experimentally.

3.1. 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. Future events are sorted by time. The basic execution cycle involves removing events from 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 future event list.

3.2. Multiple, Synchronized Event Lists

The multiple, synchronized event list execution environment is a simple extension of the basic event list mechanism for execution on a multiprocessor. In this mechanism, each processor has its own future event list. In addition, one processor has special status and acts as a global scheduler. The basic execution cycle is somewhat more complex in order to guarantee correct execution on the multiprocessor.

First, each processor sends a message to the scheduler indicating the simulation time of the next event on its event list. The scheduler selects the minimum next event time and sends a message to all the processors containing this value. Each processor having this minimum value removes events from its event list, forms event combinations, and invokes the appropriate logical processes’ actions. When an action causes an output event, that event is either inserted into the local future event list, or a message is sent to a remote processor requesting that it insert an event into its future event list. 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. Finally, the processor waits until it receives a completion message from all its predecessors. At this point the execution cycle is complete and may begin again.

In this mechanism, each logical process is statically assigned to a processor. This assignment is specified in the Yaddes source. Since the assignment is static, each process knows a priori whether to insert output events into its own future event list or to send a message to another processor.

3.3. Chandy-Misra Distributed Discrete Event Simulation

In the Chandy-Misra distributed discrete event simulation execution environment each logical process runs as a separate task on a separate processor. In this environment, the logical processes are called Envelopes. A model instantiation is associated with each envelope. Envelopes exchange messages containing events.

The basic execution cycle begins when an envelope receives a message. The envelope buffers messages until an event combination can be formed. (An event combination with simulation time \( t \) can only be formed when an envelope has received an event message for each input of its associated model having time \( t \geq t_i \).) When an event combination is formed, the appropriate action is invoked. When an action causes an output event, it causes its envelope to send event messages to the envelopes of the appropriate logical processes.
This execution environment has the potential for deadlock. We have not yet implemented any deadlock detection/recovery scheme. We require the simulation programmer to explicitly avoid deadlock.

3.4. Virtual Time based Distributed Discrete Event Simulation

The virtual time based distributed discrete event simulation mechanism is based on the time warp operating system. As in the Chandy-Misra mechanism, each logical process runs as a separate task on a separate processor and is called an Envelope. A model instantiation is associated with each envelope and envelopes exchange messages containing events.

The basic execution cycle begins when an envelope receives a message. When a message arrives, there are two possibilities — its time stamp is either before or after the current (local) value of simulation time. If its time stamp is after the current time, an input event combination is formed and the appropriate action is invoked. If its time stamp is before the current time, the envelope backs up to the time on 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 antimesages.

4. Example: Exclusive-OR Circuit

An example to illustrate the appearance of a Yaddes specification is given below. The system in this example is a logic network consisting of four NAND gates that implement the exclusive-OR function as shown in Fig. 1.

```
#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;
  }
  initial state { 0, 0 }
  action initial
  {
    OUTPUT ($out, $time + GATE_DELAY, NAND ($state->input0,
```
$state->input1));
}
action in0
{
  $state->input0 = $event [$in0];
  OUTPUT ($out, $time + GATE_DELAY, NAND ($state->input0,
  $state->input1));
}
action in1
{
  $state->input1 = $event [$in1];
  OUTPUT ($out, $time + GATE_DELAY, NAND ($state->input0,
  $state->input1));
}
action in0, in1
{
  $state->input0 = $event [$in0];
  $state->input1 = $event [$in1];
  OUTPUT ($out, $time + GATE_DELAY, NAND ($state->input0,
  $state->input1));
}
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\n", $name);
      (void) exit (1);
    }
    while (fscanf ($state->ifp, "%d%d", &time, &event) == 2)
    {
      OUTPUT ($out, time, event);
    }
  }
end model

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

process X : ReadFromFile
process Y : ReadFromFile
process Gate0 : TwoInputNand
process Gate1 : TwoInputNand
process Gate2 : TwoInputNand
process Gate3 : TwoInputNand
process Z : WriteToFile

connect X.out to Gate0.in0, Gate1.in0
connect Y.out to Gate0.in1, Gate2.in1
connect Gate0.out to Gate1.in1, Gate2.in0
connect Gate1.out to Gate3.in0
connect Gate2.out to Gate3.in1
connect Gate3.out to Z.in

The model called TwoInputNand describes the behavior of a NAND gate. TwoInputNand has two inputs called in0 and in1 and a single output called out. TwoInputNand has two state variables called input0 and input1. These variables record the logic level (0 or 1) on the corresponding inputs.

The TwoInputNand model has an initial action and three event combination actions. The event combinations are (i) an event occurred on input in0, (ii) an event occurred on input in1, and (iii) simultaneous events occurred on both in0 and in1. Each of these actions updates the appropriate fields of the state and generates an output event representing the logical NAND of the inputs after a gate delay of 10 time units.

The model called ReadFromFile represents a process that reads input events for the simulation from a file. The file is assumed to contain pairs of integers. The first integer represents the time of an event and the second integer is either 0 or 1 representing a logic level. The ReadFromFile model only has an initial action. It opens a file whose name is the same as that of the logical process executing the model and then reads input events until the end-of-file is reached.

The model called WriteToFile represents a process that writes output events from the simulation into a file. The file format is the same as the input file format described above. In its initial action the WriteToFile model opens a file whose name is the same as that of the logical process.
executing the model. The WriteToFile model has a single input. Whenever an input event arrives, the WriteToFile model prints a pair of integers representing the time and event into the output file.

A total of seven logical processes are declared in Fig. 2. In addition to the four NAND gates, there are two instances of the ReadFromFile model and an instance of the WriteToFile model.

5. The Yaddes Specification Language

The general Yaddes specification file contains specifications for models, processes, connections, and procedures. In addition, the file may also contain C language code. The format of a specification file is

```
optional C code
%%
model specifications
process specifications
connection specifications
procedure specifications
%%
optional C code
```

Note that the C language code is separate from Yaddes specifications by the symbols %%%. The C code which follows the the Yaddes specifications is optional and if omitted the second %% may also be omitted.

The procedure specifications are optional. In order for there to be any connection specifications, there must be at least one process specification. In order for there to be any process specifications, there must be at least one model specification.

5.1. Model Specifications

A model specification describes a generic logical process. Actual processes are instantiations of models. For example, if the system being simulated is a logic circuit, then the models might be describe and, or, and not gates. The logic circuit can be constructed by instantiating these models to represent the actual gates in the circuit.

The format of a model specification is

```
model model name
input statement
output statement
state statement
initial state statement
action statements
end model
```

All the statements are mandatory except for the action statements. Hence the minimal model looks like

```
model model name
input statement
output statement
state statement
initial state statement
end model
```
5.1.1. Inputs

The \texttt{inputs} statement is used to specify the number and names of the inputs of a model. A model can have zero or more inputs (up to a maximum of 30 on the VAX). Each input and output of a given model must have a distinct name. (The names of inputs and outputs of different models need not be distinct.)

For example, the statement

\begin{verbatim}
inputs a, b, c
\end{verbatim}

specifies that this model has three inputs named \texttt{a}, \texttt{b}, and \texttt{c}. The statement

\begin{verbatim}
inputs none
\end{verbatim}

specifies that this model has no inputs. (I.e., it is a source of events.)

5.1.2. Outputs

The \texttt{outputs} statement is used to specify the number and names of the outputs of a model. A model can have zero or more outputs. (There is no inherent limit on the number of outputs that a model can have.) Each input and output of a given model must have a distinct name. (The names of inputs and outputs of different models need not be distinct.)

For example, the statement

\begin{verbatim}
outputs a, b, c
\end{verbatim}

specifies that this model has three outputs named \texttt{a}, \texttt{b}, and \texttt{c}. The statement

\begin{verbatim}
outputs none
\end{verbatim}

specifies that this model has no outputs. (I.e., it is a sink of events.)

5.1.3. State

The \texttt{state} statement is used to declare the type of the state of a model. Each instantiation of a model (i.e., process) has its own unique state with the specified type. The syntax of this type declaration is that of a C language \texttt{struct} declaration.

For example, the statement

\begin{verbatim}
state
{
    int x;
    int y;
    int z;
}
\end{verbatim}

specifies that the state of the model consists of three integers \texttt{x}, \texttt{y}, and \texttt{z}. Any valid C language structure field declaration is allowed between the braces. (Due to a quirk of C, empty \texttt{struct}s are not allowed. Thus, the model state type must have at least one field.)

5.1.4. Initial State

The \texttt{initial state} field is used to specify the value to which the state of each instantiation of this model is initialized. The syntax of this statement is that of a C language structure initializer. For example, the statement
initial state \{ 0, 0, 0 \}
specifies that the first three field of the state are initialized to zero. Any valid C language structure field value is allowed between the braces. (Note that the C language compiler does not check that the number of values between the braces matches the number fields in the corresponding structure type declaration. The programmer must exercise care when initializing structures.)

5.1.5. Actions

Actions are C language program fragments that are invoked during the course of a simulation. Actions can be triggered in a number of ways. Specifically, actions can be triggered before simulation begins, before simulation ends, whenever simulated time advances, and in response to combinations of events.

5.1.5.1. General Event Combination Actions

A combination of events is a set of events that occur at the same simulation time. For example, the statement

\[
\begin{align*}
\text{action } a, b, c \\
& \{
& \quad \text{printf } ("hello world"); \\
& \}
\end{align*}
\]

specifies that whenever events occur on inputs \( a, b, \) and \( c \) at the same simulation time, the C language print statement is invoked.

The syntax of the action statement is that of a C language statement list. Any valid C language statement sequence is allowed between the braces. In addition, certain special variables and constants are available for use in the statement sequence. (See the section on predefined variables and constants.)

The same C language statements can be invoked as the result of several different event combinations. For example, the statement

\[
\begin{align*}
\text{action } a \\
\text{action } b \\
\text{action } c \\
& \{
& \quad \text{printf } ("hello world"); \\
& \}
\end{align*}
\]

specifies that whenever an event occurs on input \( a, b, \) or \( c \), the C language print statement is invoked.

5.1.5.2. Initial Action

The \texttt{action initial} statement is used to specify an action that is invoked exactly once at simulation time zero. For example, the statement

\[
\begin{align*}
\text{action initial} \\
& \{
& \quad \text{printf } ("hello world"); \\
& \}
\end{align*}
\]

specifies that the C language print statement is to be invoked at simulation time zero.

PLEASE NOTE: The initial action occurs at time 0. Since no two actions can be invoked with the same time, and since ALL processes are given the opportunity to perform an initial action, no events may be scheduled at time 0.
5.1.5.3. Final Action

The `action final` statement is used to specify an action that is invoked exactly once immediately prior to the end of the simulation. For example, the statement

```plaintext
action final
{
    printf("hello world");
}
```

specifies that the C language print statement is to be invoked immediately prior to ending the simulation.

5.1.5.4. No Event Action

The `action none` statement is used to specify an action that is invoked when no events have occurred, but simulation time has advanced. This situation will only occur when using the special function `IGNORE`. For example, the statement

```plaintext
action none
{
    printf("hello world");
}
```

specifies that the C language print statement is to be invoked whenever no events have occurred, but simulation time has advanced. (See the section on using `IGNORE`).

5.1.5.5. Default Action

The `action default` statement is used to specify an action that is invoked when an event combination has occurred for which no action has been explicitly declared. For example, the statement

```plaintext
action default
{
    printf("hello world");
}
```

specifies that the C language print statement is to be invoked when an event combination has occurred for which no action has been explicitly declared.

If a default action is not specified, a default default action is invoked. The default default action simply prints a warning message that specifies the event combination. (Note, warnings can be suppressed. See the section on running the simulation.)

5.2. Process Specifications

A `process` statement declares an instantiation of a model. Each such instantiation executes as a separate logical process. Each such process has its own distinct state. For example, the statements

```plaintext
process Gate1 : Nand
process Gate2 : Nand
process Gate2 : Nand
```

specify three processes called `Gate1`, `Gate2`, and `Gate3`. They are all instances of the model called `Nand`. Each of these processes has its own distinct state. The type of this state is that specified in the `Nand` model. Each process has the set of inputs and a set of outputs that is specified in the `Nand` model.
In some execution environments there may be several processors on which the processes may execute. In this case, each process must be assigned to a processor. For example, the statements

\begin{verbatim}
process Gate1 on 0 : Nand
process Gate2 on 0 : Nand
process Gate2 on 1 : Nand
\end{verbatim}

specify that the processes Gate1 and Gate2 execute on processor number 0 and process Gate3 executes on processor number 1.

Processors are numbered starting with 0. There may be any number of processors. The number of processors is obtained by taking the largest processor number specified and adding 1. Note that it is possible to have a processor unused simply by not assigning processes to it. If a process is not explicitly assigned to a processor, it is assigned to processor number 0 by default.

5.3. Connection Specifications

The connect statement is used to establish connections between the inputs and outputs of logical processes. Each output of a logical process can be connected to zero, one or more inputs. (I.e., fan-out is allowed.) Each input of a logical process can be connect to at most one output. (I.e., fan-in is not allowed.) The inputs and outputs of processes are specified in the form process name . port name, where process name is the name of a process, and port name is the name of an input or output of that process. For example, the statement

\begin{verbatim}
connect Gate1.out to Gate2.in0
\end{verbatim}

specifies that the output out from the process named Gate1 is connected to the input in0 of the process named Gate2. An output port with fan-out can be specified by connecting to more than one input port. For example, the statement

\begin{verbatim}
connect Gate1.out to Gate2.in0, Gate2.in0
\end{verbatim}

specifies a fan-out of two.

5.4. Procedure Specifications

5.4.1. Initial procedure

The initial procedure declaration allows the Yaddes user to specify the name of a C subroutine that is to be called before the simulation begins. For example, the statement

\begin{verbatim}
procedure SetUp : initial
\end{verbatim}

specifies that the name of the C language subroutine that is to be invoked before the simulation begins is SetUp. It is the responsibility of the Yaddes user to provide this routine.

5.4.2. Final procedure

The final procedure declaration allows the Yaddes user to specify the name of a C subroutine that is to be called after the simulation ends. For example, the statement

\begin{verbatim}
procedure CleanUp : final
\end{verbatim}

specifies that the name of the C language subroutine that is to be invoked after the simulation ends is CleanUp. It is the responsibility of the Yaddes user to provide this routine.
5.5. Predefined Variables and Constants

The Yaddes user has access to certain predefined variables and constants inside the C language portions of actions. The names of these variables and constants all begin with the symbol \$.

5.5.1. Constants

Each of the inputs and outputs of a model is assigned an integer value. For example, if a model has the following input specification

\[ \text{inputs a, b, c} \]

then the constants \$a, \$b, and \$c are defined. (In the current version of Yaddes the values will be 2, 1, and 0, respectively.)

Similarly, if a model has the following output specification

\[ \text{outputs a, b, c} \]

then the constants \$a, \$b, and \$c are defined. (In the current version of Yaddes the values will be 2, 1, and 0, respectively.)

For information on the use of these values, see the section on predefined functions.

5.5.2. Variables

5.5.2.1. \$event

The \$event variable is an array of integers. Its declaration is equivalent to the following:

\[ \text{int \$event [];} \]

When an action is invoked, the event array contains the events that triggered that action. For example, if the action trigger is

\[ \text{action a, b} \]

then \$event[$a] contains the event that occurred on input a and \$event[$b] contains the event that occurred on input b. If the model has a third input, c say, then the value in \$event[$c] is undefined. (In the current version undefined values are always 0. This may not be the case in future versions of Yaddes. The Yaddes user should use the predefined function EVENT_OCCURRED to determine if the \$event array entry is defined for a particular input.)

5.5.2.2. \$name

The \$name variable is a pointer to a character string that contains the name of the process (not of the model). Its declaration is equivalent to the following:

\[ \text{char * \$name;} \]

This can be used in a print statement to aid debugging.

5.5.2.3. \$state

The \$state variable is a pointer to a structure whose type is that declared in the \text{state} statement of the model specification. Using this pointer, the Yaddes user can alter the current state of a process.
5.5.2.4. \$time

The \$time variable is an integer. Its declaration is equivalent to the following:

```c
int \$time;
```

It contains the current value of simulation time.

5.6. Predefined Functions

The Yaddes user has access to certain predefined functions inside the C language portions of actions. These functions are used to cause events to occur on the outputs of processes and to control the simulation.

5.6.1. ACTIVATE

The ACTIVATE function is used in conjunction with the DEACTIVATE function. The declaration of this function is equivalent to the following:

```c
void ACTIVATE (port)
int port;
```

The purpose of this function is to turn on an input to a process that has been previously turned off by the DEACTIVATE function. For example, the statement

```c
ACTIVATE ($a);
```

turns input \( a \) on.

5.6.2. DEACTIVATE

The DEACTIVATE function is used to turn off an input to a process. The declaration of this function is equivalent to the following:

```c
void DEACTIVATE (port)
int port;
```

For example, the statement

```c
DEACTIVATE ($a);
```

turns off input \( a \). When an input is turned off, it means that no events are expected on that input. Should any events occur on a DEACTIVATEd input, warning messages will be printed (unless warnings are suppressed). (See the section on running the simulation.)

5.6.3. EVENT_OCCURRED

The EVENT_OCCURRED function is used to test whether an input has an event on it. The declaration of this function is equivalent to the following:

```c
int EVENT_OCCURRED (port)
int port;
```

This function returns zero if no event has occurred on the specified input. It returns a nonzero result if an event has occurred.

This function is useful in actions associated with multiple event combinations. The following example illustrates the use of this function in a multiple event combination action:
action a
action a, b
{
    if (EVENT_OCCURRED ($b))
        printf ("hello world");
}

5.6.4. IGNORE

The IGNORE function is used to cause an input to a process to be turned off until simulation time has advanced past a specified time value. The declaration of this function is equivalent to the following:

void IGNORE (port, time)
int port;
int time;

For example, the statement

IGNORE ($a, $time + 10);

specifies that input $a is to be turned off until simulation time has advanced to the current simulation time plus ten time units. When an input is turned off, it means that no events are expected on that input. Should any events occur on an IGNOREd input, warning messages will be printed (unless warnings are suppressed). (See the section on running the simulation.)

5.6.5. OUTPUT

The OUTPUT function is used to cause an event to occur on an output of a process. The declaration of this function is equivalent to the following:

void OUTPUT (port, time, event)
int port;
int time;
int event;

For example, the statement

OUTPUT ($a, $time + 10, 57);

specifies that an event is to occur on output $a at current simulation time plus ten time units. The event value is 57.

5.6.6. NULL_OUTPUT

The NULL_OUTPUT function is similar to the OUTPUT function. Its declaration is equivalent to the following:

void NULL_OUTPUT (port, time, event)
int port;
int time;
int event;

This function is used in the same manner as OUTPUT to cause events to occur on the outputs of a process. This semantics of this function are identical to those of the OUTPUT function. Its purpose is to identify those events that are required to avoid deadlock when using the Chandy-Misra distributed discrete event simulation mechanism. Under other execution mechanisms, NULL_OUTPUT is a no-op.
5.6.7. SET_TIME_LIMIT

The SET_TIME_LIMIT function is used to specify the simulation time at which a process is to terminate. The declaration of this function is equivalent to the following:

```c
void SET_TIME_LIMIT (time)
int time;
```

For example, the statement

```c
SET_TIME_LIMIT (1000);
```

will cause the process to terminate execution when simulated time advances past time 1000.

This function is usually called in the initial action of a model. Note that when a process terminates, its final action is invoked. In effect, this function schedules the final action at a specified time.

The precise semantics of this function depends on the execution mechanism used. In some execution environments, each process has its own notion of time. In this case, SET_TIME_LIMIT only affects the process in which it is invoked. In execution environments using global time synchronization, a global time limit is used. In this case, the time limit is the minimum value of time specified in all of the calls to SET_TIME_LIMIT.

5.7. Compiling and Running the Simulation

5.7.1. Compiling a Yaddes Language Specification

The Yaddes program transforms a specification file into a C language file. The Yaddes program can be invoked in two ways. For example,

```bash
yaddes input file
```

transforms the specifications in the input file. If the input file name ends in .y, the output file name is constructed by replacing the .y with .c. Otherwise .c is added to the end of the input file name to construct the output file name.

If Yaddes is invoked without an argument, it reads the standard input file and writes the standard output file. For example,

```bash
yaddes < input file > output file
```

5.7.1.1. Separate Compilation

If a simulation contains many models, it may become convenient to separate the models into different input files. Yaddes supports a primitive form of separate compilation in that model specifications can appear in separate files. However, all the process, connection, and procedure declarations must appear in one file.

Since processes are instantiations of models, Yaddes needs some information in the file containing the process specifications about the models being instantiated. Yaddes supports external model specifications which look like the following:
Note that an external model specification contains no action specifications. The external model specification specifies that a model with the given inputs, output, state, and initial state is declared in a different file. It is up to the Yaddes user to ensure that the external model specification and the actual model specification agree. An easy error to commit (and difficult to detect and correct) is to change the model specification in one file without changing the external model specification in another file. Be forewarned.

5.7.1.2. Run-time Execution Support Mechanism Libraries

Once the C language program file (or files) has been produced, the C compiler can be used to create an executable object module. In order to create this module, the C files must be linked to to an execution mechanism run-time support library. There are currently eight such libraries available. The libraries are summarized in the following table:

<table>
<thead>
<tr>
<th>library</th>
<th>description</th>
<th>portable</th>
</tr>
</thead>
<tbody>
<tr>
<td>event-list.a</td>
<td>traditional event-list driven simulation kernel</td>
<td>yes</td>
</tr>
<tr>
<td>multi-list.a</td>
<td>multiple, synchronized event-lists kernel</td>
<td>yes</td>
</tr>
<tr>
<td>chandy-misra.a</td>
<td>Ch Andy-Misra distributed discrete event simulation kernel</td>
<td>yes</td>
</tr>
<tr>
<td>virtual-time.a</td>
<td>virtual-time-based distributed discrete event simulation kernel</td>
<td>yes</td>
</tr>
<tr>
<td>multi-list.vax.a</td>
<td>multiple, synchronized event-lists kernel that simulates execution on a multiprocessor by multitasking under a single unix process</td>
<td>VAX only</td>
</tr>
<tr>
<td>chandy-misra.vax.a</td>
<td>Chandy-Misra distributed discrete event simulation kernel that simulates execution on a multiprocessor by multitasking under a single unix process</td>
<td>VAX only</td>
</tr>
<tr>
<td>virtual-time.vax.a</td>
<td>virtual-time-based distributed discrete event simulation kernel that simulates execution on a multiprocessor by multitasking under a single unix process</td>
<td>VAX only</td>
</tr>
</tbody>
</table>

For example, the sequence of commands

```bash
yaddes example.y
cc -o example example.c yaddes.a
```
will produce an executable object module called \texttt{example} from the Yaddes Language specification file \texttt{example.y}.

5.7.2. Running a Simulation Program

The run-time execution mechanism libraries support a number of command line arguments. These arguments turn on or off various the printing of various statistics and debugging messages. All messages are printed on the standard error file. If the executable module is called \texttt{example} then the format of the command to invoke the simulation is

\texttt{example [flag [flag ...]]}

Note that one or more flags may be included on the command line.

5.7.2.1. +a

This flag turns on all forms of output. (Except the help output.)

5.7.2.2. +h and −h

These flags cause a brief summary of the flags to be printed. The simulation is not run when this flag is specified.

5.7.2.3. +i

This flag causes the printing of messages that trace calls to \texttt{IGNORE}, \texttt{ACTIVATE}, and \texttt{DEACTIVATE}.

5.7.2.4. +k

This flag causes the printing of messages that trace the execution of the kernel.

5.7.2.5. +m

This flag causes the printing of a summary of message statistics. This summary specifies the number of messages received on each input of every process.

5.7.2.6. +o

This flag causes the printing of messages that trace calls to \texttt{OUTPUT} and \texttt{NULL_OUTPUT}.

5.7.2.7. +p

This flag causes a printing of a post mortem when the simulation terminates. The post mortem lists all the message that have been received by processes, but not yet processed.

5.7.2.8. +s

This flag causes the printing of various statistics gathered by the kernel. These statistics include the number of messages sent and received, the number of actions invoked, and the number of context switches. The statistics produced by the various kernels are described in the appendices.

5.7.2.9. +t

This flag causes the printing of a detailed trace of actions as they are invoked.
5.7.2.10. \(-w\)

This flag suppress the printing of warning messages.

6. Pseudo-Random Number Generator Package

The Yaddes system provides support for multiple, independent pseudo random number streams. This allows each processes to have its own stream of random numbers that is independent of the other processes. This is essential in order to ensure that the results of the simulation are independent of the underlying simulation mechanism.

6.1. Using the Pseudo-Random Number Generator Package

Pseudo-random number generators require internal state in order to operate. Every time a random number is generated, the internal state is modified. In order to ensure correct operation in the Yaddes environment, this state must be part of the state of the process. The Yaddes system provides a predefined type, `RNG_TYPE`, and initial value, `RNG_INITIAL_VALUE`, that are to be used in the model state and initial state specifications as shown here:

```c
state
{
    RNG_TYPE rng_state;
    ...
}
initial state
{
    RNG_INITIAL_VALUE,
    ...
}
```

Pseudo-random number generations must be initialized before use. The `InitializeRNG` routine is used to initialize the pseudo-random number generator. The declaration of this routine is equivalent to the following:

```c
void InitializeRNG (seed, state_ptr)
int seed;
RNG_TYPE *state_ptr;
```

The first argument is an integer used to seed the random number generator. The second argument is a pointer to a random number generator state variable. A convenient place to initialize the pseudo-random number generator in Yaddes is in an initial action as shown here:

```c
action initial
{
    InitializeRNG (1, &($state->rng_state));
}
```

If the call to this routine is omitted, then the default initial state, `RNG_INITIAL_STATE`, is used. This state is equivalent to initializing using a seed of 1.

The routine called `Random` is used to generate the actual pseudo-random number sequence. The declaration of this routine is equivalent to the following:

```c
int Random (state_ptr)
RNG_TYPE *state_ptr;
```
Every time this routine is invoked it returns a different integer in the range 0 to $2^{31}-1$. The following statement shows how to invoke the pseudo-random number generator:

\[ n = \text{Random} \,(\&\,(\text{state} \rightarrow \text{rng\_state})); \]

7. Statistics Gathering and Reporting Package

The Yaddes system provides a set of routines that facilitate the gathering and reporting of statistics. Two types of statistic are provided: population and time averaged. The reporting routines report for each statistic the minimum value, the maximum value, the number of values, the sum of the values, the sum of the squares of the values, the mean, and the standard deviation.

In order to accumulate statistics, they must be incorporated into the state of the process. The Yaddes system provides a predefined type, \texttt{STATISTIC\_TYPE}, and initial value, \texttt{STATISTIC\_INITIAL\_VALUE}, that are to be used in the model state and initial state specifications as shown here:

```c
state
{  
    \texttt{STATISTIC\_TYPE} stat1;
    \texttt{STATISTIC\_TYPE} stat2;
    ...
}
initial state
{  
    \texttt{STATISTIC\_INITIAL\_VALUE},
    \texttt{STATISTIC\_INITIAL\_VALUE},
    ...
}
```

Note that each process may accumulate more than one statistic.

The Yaddes user must initialize the state of each statistic and declare whether the statistic is a population or time-averaged statistic. The routine \texttt{DeclareStatistic} is used to initialize a statistic. The declaration of this routine is equivalent to the following:

```c
void DeclareStatistic (stat_ptr, type)
    \texttt{STATISTIC\_TYPE} *stat_ptr;
    int type;
```

The first argument is a pointer to a statistic state variable. The second argument specifies the type of the statistic. There are two types currently supported: \texttt{POPULATION} and \texttt{TIME\_AVERAGED}. A convenient place to initialize statistics in Yaddes is in an initial action as shown here:

```c
action initial
{  
    DeclareStatistic (\&($state\rightarrow stat1), \texttt{POPULATION});
    DeclareStatistic (\&($state\rightarrow stat2), \texttt{TIME\_AVERAGED});
}
```

If the call to this routine is omitted, then the default initial state, \texttt{STATISTIC\_INITIAL\_STATE}, is used. This state is equivalent to declaring the statistic to be a population statistic.

The routine \texttt{AccumulateStatistic} is used to gather statistics. The declaration of this routine is equivalent to the following:
void AccumulateStatistic (stat_ptr, value, time)
STATISTIC_TYPE *stat_ptr;
int value;
int time;

The first argument is a pointer to a statistic state variable. The second argument is the new value of the statistic. The third argument is required to be the current value of simulation time. The following statement shows how to accumulate a statistic:

    int val;
    val = ... ;
    AccumulateStatistic (&($state->stat1), val, $time);

The routine called ReportStatistic is used to print a summary of a statistic on the standard output file. The declaration of this routine is equivalent to the following:

    void ReportStatistic (stat_ptr, label);
    STATISTIC_TYPE *stat_ptr;
    char *label;

The first argument is a pointer to a statistic state variable. The second argument is a pointer to a character string to be used to label the output. A convenient place to report statistics in Yaddes is in a final action as shown here:

    action final
    {
        ReportStatistic (&($state->stat1), "customer age");
        ReportStatistic (&($state->stat2), "customers in queue");
    }

8. Avoiding deadlock under the Chandy-Misra execution mechanism

A problem with the Chandy-Misra execution environment is the possibility for deadlock. To understand how deadlock can arise, consider the system of processes shown schematically in Fig. 2. The system consists of two processes—X and Y. Assume that process Y only outputs events on its output arc in response to an event arriving on its input arc. This means that process X only receives input events from Y if it has sent an event to Y at some earlier time. The problem is that the execution environment does not know that no event will arrive. Since event combinations must be presented to process X in time stamp order, the execution environment is deadlocked.

![Fig. 2.](image)

In the following sections we introduce three different ways to avoid deadlock in the Yaddes implementation of the Chandy-Misra execution environment. The Yaddes implementation currently does not support deadlock detection and recovery mechanisms. Hence, the simulation programmer must use one of the following methods to avoid deadlock.
8.1. Using OUTPUT and NULL_OUTPUT

One way to avoid deadlock is to emit events that do not correspond to events in the real-world system. Such null events carry no information, but are merely there to avoid deadlock. For example, in Fig. 2., process X could send a null event to Y indicating that no output is present at time t. Process Y then responds with a null event back to X. Process X may then proceed since it now knows that no event will arrive from Y.

In the Yaddes system, null events can be emitted using the NULL_OUTPUT routine. The arguments to this routine are identical to those of the OUTPUT routine. The NULL_OUTPUT routine merely identifies the events in a simulation that are to avoid deadlock. Since the other execution mechanisms do not suffer from deadlock, the NULL_OUTPUT function has no effect under the other mechanisms.

8.2. Using IGNORE

A second way to avoid deadlock is to have a process announce that it is not expecting to receive events on a given input. For example, in Fig. 2., it may be that process X knows that it will not receive any events from process Y until after some time t. The IGNORE function can be used to inform the execution environment that no events will arrive from Y until after time t. Since execution mechanisms other than Chandy-Misra do not suffer from mechanisms.

8.3. Using ACTIVATE and DEACTIVATE

The third way to avoid deadlock is a variant of the previous method. Again, a process announces that it is not expecting to receive events on a given input. The DEACTIVATE function is used to inform the execution environment that, until further notice, no input events will arrive on a given input. The ACTIVATE function is used to turn an input back on after it has been deactivated. Whereas in the case of the IGNORE function, the process must know a future time after which events may arrive, in the case of ACTIVATE and DEACTIVATE the process explicitly reactivates at some future time. Both of these approaches has its disadvantages: The IGNORE routine requires a process to predict the future whereas the DEACTIVATE routine requires a process to remember that an input has been deactivated.

9. Example: Queue and Server

This example illustrates the use of Yaddes for constructing a simple queueing system simulation. This example illustrates the use of the Yaddes pseudo-random number generator package and the statistics gathering and reporting package. This example also illustrates the use of Yaddes features to avoid deadlock under the Chandy-Misra run-time execution mechanism.

The queueing system being simulated in this example is shown in Fig. 3. There are three physical processes in this system—the arrival process, the queue, and the server process. The Yaddes specification of each of these processes will be given in the following sections.

![Fig. 3.](image-url)
9.1. The Arrival Process

The Yaddes specification of the arrival process is given below. A Poisson arrival process is specified, i.e., the interarrival times are exponentially distributed. The arrival process has a single input and a single output. The arrival process is a self-timed source of events, i.e., its output is connected back to its input.

The initial action of the arrival process initializes the random number generator and a statistic that will keep track of the interarrival time. The initial action then selects a random interarrival time from an exponential distribution and outputs an event with that time. This event represents the arrival of a customer.

Whenever an event arrives at the input to the arrival process, it selects a new random interarrival time from an exponential distribution and outputs an event with that time. Since the output of the Arrival process is connected back to itself, it is a self-timed source of events.
model Arrival
    inputs in
    outputs out
    state
    {
        STATISTIC_TYPE arrivalTime;
        RNG_TYPE producerRNG;
    }
    initial state { STATISTIC_INITIAL_VALUE, RNG_INITIAL_VALUE }
    action initial
    {
        int arrTim;

        InitializeRNG (SEED, &($state->producerRNG));
        DeclareStatistic (&($state->arrivalTime), POPULATION);
        arrTim = ExpDistRandVar (MEAN_ARRIVAL_TIME,
            &($state->producerRNG));
        AccumulateStatistic (&($state->arrivalTime), arrTim, $time);
        OUTPUT ($out, $time + arrTim, 0);
        SET_TIME_LIMIT (10000);
    }
    action in
    {
        int arrTim;

        arrTim = ExpDistRandVar (MEAN_ARRIVAL_TIME,
            &($state->producerRNG));
        AccumulateStatistic (&($state->arrivalTime), arrTim, $time);
        OUTPUT ($out, $time + arrTim, 0);
    }
    action final
    {
        ReportStatistic (&($state->arrivalTime),
            "inter arrival time");
    }
end model

9.2. The Queue Process

The Yaddes specification of the queue process is given below. The queue process has two inputs
and one output. Input events from the arrival process represent the arrival of a customer. Input events
from the server process represent a request for the next customer in the queue.

The initial action of the queue process initializes two statistics: one to keep track of the (time-
averaged) queue length, and one to keep track of the the time spent customers in the queue. The initial
action also deactivates the request input. This is needed to avoid deadlock. The protocol is that the server
will only issue a request after it has finished servicing a customer. Since no customers have been sent to
the server, no request will arrive. When the request input is deactivated, the queue process sets a Boolean
value in its state to indicate that a request is pending. This means that a customer can be forwarded
directly to the server as soon as it arrives. Before forwarding the customer, the queue must reactivate the
request input so that it can receive the next request from the server.
There are three possible event combinations that the queue process must deal with: 1. the arrival of a customer, 2. the arrival of a request, and 3. the simultaneous arrival of an event and a customer. Generally, when a customer arrives, the queue process enqueues the arrival time of that customer. When a request arrives, the first arrival time is dequeued. At each event, the queue process updates the appropriate statistics.

```plaintext
model Queue
  inputs in, request
  outputs out
  state
  {
    STATISTIC_TYPE queueLength;
    STATISTIC_TYPE timeInQueue;
    int requestPending;
    int head;
    int tail;
    int count;
    int queue [MAX_QUEUE_LENGTH];
    int queueTime [MAX_QUEUE_LENGTH];
  }
initial state
  {
    STATISTIC_INITIAL_VALUE,
    STATISTIC_INITIAL_VALUE,
    1,
    0,
    0,
    0
  }
action initial
  {
    DeclareStatistic (&($state->queueLength), TIME_AVERAGED);
    DeclareStatistic (&($state->timeInQueue), POPULATION);
    DEACTIVATE ($request);
    SET_TIME_LIMIT (10000);
  }
action in
  {
    if ($state->requestPending)
    {
      $state->requestPending = 0;
      ACTIVATE ($request);
      OUTPUT ($out, $time, $event [$in]);
    }
    else
    {
      if ($state->tail == MAX_QUEUE_LENGTH - 1)
        $state->tail = 0;
      else
        $state->tail += 1;
    }
  }
```

Copyright (c) 1988 by Bruno R. Preiss. All rights Reserved.
if ($state->tail == $state->head)
{
    (void) fprintf (stderr,
        "\%s: queue overflow - client discarded\n", $name);
} else
{
    $state->queue [$state->tail] = $event [$in];
    $state->queueTime [$state->tail] = $time;
    AccumulateStatistic (&($state->queueLength), $state->count, $time);
    $state->count += 1;
}
}
action request
{
    int waitTime;
    if ($state->head == $state->tail)
    {
        DEACTIVATE ($request);
        $state->requestPending = 1;
    } else
    {
        if ($state->head == MAX_QUEUE_LENGTH - 1)
            $state->head = 0;
        else
            $state->head += 1;
        OUTPUT ($out, $time, $event [$in]);
        waitTime = $time - $state->queueTime [$state->head];
        AccumulateStatistic (&($state->timeInQueue), waitTime, 0);
        $state->count -= 1;
    }
}
action in, request
{
    int waitTime;
    if ($state->head == $state->tail)
    {
        OUTPUT ($out, $time, $event [$in]);
    } else
    {
        if ($state->head == MAX_QUEUE_LENGTH - 1)
            $state->head = 0;
        else
            $state->head += 1;
        $state->queue [$state->head] = $event [$in];
        $state->queueTime [$state->head] = $time;
        AccumulateStatistic (&($state->queueLength), $state->count, $time);
        $state->count += 1;
    }
}
else
    $state->head += 1;
OUTPUT ($out, $time, $state->queue [$state->head]);
waitTime = $time - $state->queueTime [$state->head];
AccumulateStatistic (&($state->timeInQueue), waitTime, 0);
if ($state->tail == MAX_QUEUE_LENGTH - 1)
    $state->tail = 0;
else
    $state->tail += 1;
$state->queue [$state->tail] = $event [$in];
$state->queueTime [$state->tail] = $time;
}
action final
{
    AccumulateStatistic (&($state->queueLength), $state->count, $time);
    ReportStatistic (&($state->queueLength), "queue length");
    ReportStatistic (&($state->timeInQueue), "time in queue");
    (void) printf ("Queue: %d clients left in queue at end of simulation\n", $state->count);
}
end model

9.3. The Server Process

The Yaddes specification of the server process is shown below. The server process has one input and two outputs. The input to the server process represents the arrival of customers. The outputs of the server process represent the departure of customers and the issuing of request for new customers.

The initial action of the server initializes the pseudo-random number generator and a statistic that will keep track of the service times.

Whenever a customer arrives at the server, a service time is selected from an exponential distribution. Then, an output event is scheduled representing the departure of the customer, and a request for a new customer is sent to the queue process.
model Server
  inputs in
  outputs request, out
  state
  {
    STATISTIC_TYPE serviceTime;
    RNG_TYPE serverRNG;
  }
  initial state { STATISTIC_INITIAL_VALUE, RNG_INITIAL_VALUE }
  action initial
  {
    InitializeRNG (SEED, &($state->serverRNG));
    DeclareStatistic (&($state->serviceTime), POPULATION);
    SET_TIME_LIMIT (10000);
  }
  action in
  {
    int servTim;

    servTim = ExpDistRandVar (MEAN_SERVICE_TIME,
      &($state->serverRNG));
    OUTPUT ($out, $time + servTim, $event [$in]);
    OUTPUT ($request, $time + servTim, 0);
    AccumulateStatistic (&($state->serviceTime), servTim,
      $time);
  }
  action final
  {
    ReportStatistic (&($state->serviceTime), "service time");
  }
end model

9.4. The complete specification

In order to complete the Yaddes specification, the processes and connections must be declared. (In
addition, some C housekeeping must be done.) The complete specification of the Queue and Server exam-
ple is shown below.

```
#include <stdio.h>
#include <math.h>

#define MAX_QUEUE_LENGTH 32
#define MEAN_ARRIVAL_TIME 200
#define MEAN_SERVICE_TIME 200
#define SEED 1

extern int ExpDistRandVar();
```
The arrival model specification goes here.

The queue model specification goes here.

The server model specification goes here.

process arrival : Arrival
process queue : Queue
process server : Server

connect arrival .out to arrival .in, queue .in
connect queue .out to server .in
connect server .request to queue .request

%%

int ExpDistRandVar (mean, state)
    int mean;
    RNG_TYPE *state;
{
    long randLong;
    int result;

do
{
    do
    {
        randLong = Random (state);
    }
    while (randLong == 0);
    result = (int) -((double) mean * log ((double) randLong / (double) MAX_INT));
}
while (result == 0);
return (result);
}

Appendix 1. Yaddes BNF
file ::= preamble specifications postamble
preamble ::= ε \mid program
specifications ::= \% modelList processList connectionList procedureList
postamble ::= ε \mid \% program
modelList ::= ε \mid modelList model
processList ::= ε \mid processList process
connectionList ::= ε \mid connectionList connection
procedureList ::= ε \mid procedureList procedure
model ::= model identifier inputPart outputPart
statePart initialStatePart actionList end model \mid external model identifier inputPart outputPart
statePart initialStatePart end model
process ::= process identifier : identifier \mid process identifier on constant : identifier
connection ::= connect port to portList
procedure ::= procedure identifier : initial \mid procedure identifier : final
inputPart ::= inputs nameList \mid inputs none
outputPart ::= outputs nameList \mid outputs none
statePart ::= state \{ struct-decl-list \}
initialStatePart ::= initial state \{ initializer-list \}
actionList ::= ε \mid actionList action
port ::= identifier . identifier
portList ::= port \mid portList , port
nameList ::= identifier \mid nameList , identifier
action ::= actionHeaderList \{ statement-list \}
actionHeaderList ::= actionHeader \mid actionHeaderList actionHeader
actionHeader ::= action initial \mid action final \mid action default \mid action none \mid action nameList
program ::= see Kernighan & Ritchie, p. 218.
identifier ::= see Kernighan & Ritchie, p. 179.
constant ::= see Kernighan & Ritchie, p. 180.
struct-decl-list ::= see Kernighan & Ritchie, p. 216.
initializer-list ::= see Kernighan & Ritchie, p. 217.
statement-list ::= see Kernighan & Ritchie, p. 217.

Appendix 2. Kernel Statistics from event-list.a

number of events posted — The number of events inserted into the event list. Events are inserted into the event list as a result of calls to the OUTPUT routine.

number of events not posted — The number of events not inserted into the event list. Events are not inserted into the event list when the NULL_OUTPUT routine is called.
number of events processed — The number of events removed from the event list and processed. This number may be less than the number of events inserted into the event list as a result of the simulation reaching the time limit set using a call to the SET_TIME_LIMIT routine.

number of generations — A new generation is said to begin every time the simulation clock advances. The number of generations is simply the number of different values of the clock during the simulation.

average generation size — The size of a generation is defined as the number of model calls made in that generation. The average generation size is the average number of model calls over all generations.

number of model calls — The total number of model calls made during the simulation.

average model calls per generations — Exactly the same as the average generation size above.

Appendix 3. Kernel Statistics from multi-list[vax].a

number of messages sent — The total number of messages sent. This includes event messages and control messages. Event messages are sent as a result of calls to the OUTPUT routine. Control messages are used to synchronize the multiple event lists.

number of messages received — The total number of messages received. This should be exactly the same as the total number of messages sent.

number of null messages not sent — The total number of event messages not sent. Event messages are not sent when the NULL_OUTPUT routine is called. (These messages are not included in the total number of messages sent.)

number of control messages — The total number of control messages sent. Control messages are used to synchronize the multiple event lists.

number of events posted — The number of events inserted into local event lists. Events are inserted into local event lists as a result of calls to the OUTPUT routine.

number of events not posted — The number of events not inserted into local event lists. Events are not inserted into local event lists when the NULL_OUTPUT routine is called.

number of events processed — The number of events removed from the local event lists and processed. This number may be less than the number of events inserted into the local event lists as a result of the simulation reaching the time limit set using a call to the SET_TIME_LIMIT routine.

number of context switches — [vax version only] The number of context switches used in simulating the multiprocessor.

number of generations — A new generation is said to begin every time the simulation clock advances. The number of generations is simply the number of different values of the clock during the simulation.

number of model calls — The total number of model calls made during the simulation.

number of model calls per generation — The average number of model calls made in each generation.

number of envelopes — The number of processors in the simulated multiprocessor.

average busy fraction — The average fraction of processors busy in each generation. A processor is busy if it has at least one model call to make. This figure does not take account of the fact that some processors may have more than one model call to make a generation.
Appendix 4. Kernel Statistics from chandy-misra[.vax].a

number of messages sent — The total number of messages sent. This includes event messages and null messages and a message sent by the kernel to each process when deadlock is detected (e.g., at the end of the simulation). Event messages are sent as a result of calls to the OUTPUT routine. Null messages are sent as a result of calls to the NULL_OUTPUT routine.

number of messages received — The total number of messages received. This should be exactly the same as the total number of messages sent.

number of null messages — The total number of null messages sent. Null messages are sent as a result of calls to the NULL_OUTPUT routine.

number of context switches — [.vax version only] The number of context switches used in simulating the multiprocessor.

number of generations — [.vax version only] The basic cycle executed by the envelope involves receiving a message, possibly making some model calls (which may send messages), and then receiving the next message. In each generation, all envelopes are allowed to receive at most one message.

average generation size — [.vax version only] The size of a generation is the number of envelopes that process a message in that generation. The average generation size is the average over all generations of the size of a generation.

number of model calls — The total number of model calls made during the simulation.

average model calls per generation — [.vax version only] The average number of models calls in each generation.

Appendix 5. Kernel Statistics from virtual-time[.vax].a

number of context switches — [.vax version only] The number of context switches used in simulating the multiprocessor.

number of generations — [.vax version only] The basic cycle executed by the envelope involves receiving a message, possibly making some model calls (which may send messages), and then receiving the next message. In each generation, all envelopes are allowed to receive at most one message.

average generation size — [.vax version only] The size of a generation is the number of envelopes that process a message in that generation. The average generation size is the average over all generations of the size of a generation.

number of model calls — The total number of model calls made during the simulation.

average model calls per generation — [.vax version only] The average number of models calls in each generation.

number of messages sent — The total number of messages sent. This includes event messages, antimessages, and token messages. Event messages are sent as a result of calls to the OUTPUT routine. Antimessages are sent as a result of rolling back states. Token messages are used to implement fossil collection.

number of messages received — The total number of messages received. This should be exactly the same as the total number of messages sent.

number of null messages not sent — The total number of event messages not sent. Event messages are not sent when the NULL_OUTPUT routine is called. (These messages are not included in the total number of messages sent.)

number of antimessages — The total number of antimessages sent. Antimessages are sent as a result of rolling back states.
number of token messages — The total number of token messages. A circulating token is used to implement fossil collection.

number of context switches — [.vax version only] The number of context switches used in simulating the multiprocessor.

number of model calls — The total number of model calls made during the simulation.

number of states rolled back — The total number of states rolled back. This is the number of states that were erroneously entered and had to be discarded during roll back.

Appendix 6. Yaddes Compiler Output

In this appendix, the output of the Yaddes compiler will be described. The following Yaddes input file uses many of the yaddes features. The output produced by the Yaddes compiler for this input file is shown following the input file.

Yaddes Source:

```
%%
model FooBar
  inputs Alpha, Beta
  outputs Xi, Delta
  state { int Epsilon; }
  initial state { 1234 }
  action initial { SET_TIME_LIMIT (10000); }
  action Alpha { DEACTIVATE ($Beta); }
  action Beta
  action Alpha, Beta
  {
    if (EVENT_OCCURRED ($Alpha))
      OUTPUT ($Xi, $time + 10, $event [$Alpha]);
  }
  action none { IGNORE ($Alpha, $time + 20); }
  action final { }
end model

process Zero on 0 : FooBar
process One on 1 : FooBar

connect Zero .Xi to One .Alpha
connect Zero .Delta to One .Beta
connect One .Xi to Zero .Alpha
connect One .Delta to Zero .Beta

procedure InitialProcedure : initial
procedure FinalProcedure : final
```

Yaddes Compiler Output:
#line 1 "asdf"
#ifndef YADDES_DOT_H
#include "yaddes.h"
#endif

typedef struct
{
    int Epsilon;
} xxFooBarSTATE;

char *xxFooBarInputs [] = {
    "Beta",
    "Alpha",
    0
};

char *xxFooBarOutputs [] = {
    "Delta",
    "Xi",
    0
};

void FooBar (xxState, xxTime, xxMask, xxEventList,
    xxInputList, xxOutputList,
    xxInputTable, xxOutputTable,
    xxChannelTime, xxQueueHead,
    xxTimeLimit, xxActiveChannels)
    xxFooBarSTATE *xxState;
int xxTime;
int xxMask;
EVENT *xxEventList;
char **xxInputList;
char **xxOutputList;
int *xxInputTable;
int *xxOutputTable;
int *xxChannelTime;
MESSAGE_QUEUE *xxQueueHead;
int *xxTimeLimit;
int *xxActiveChannels;
{
    switch (xxMask)
    {
    case 4:
        #line 7 "asdf"
        { SET_TIME_LIMIT (10000); }
        break;
    case 2:
        #line 8 "asdf"
        { DEACTIVATE (0); }
        break;
    case 1:

51 case 3:
52  #line 11 "asdf"
53  {
54     if (EVENT_OCCURRED (1))
55         OUTPUT (1, xxTime + 10, xxEventList [1]);
56     break;
57  }
58 case 0:
59  #line 14 "asdf"
60  { IGNORE (1, xxTime + 20); }
61 break;
62 case 8:
63  #line 15 "asdf"
64  {
65  }
66 default:
67  DefaultAction (xxMask, xxEventList, xxInputList);
68  break;
69  }
70}
71
72 xxFooBarSTATE xxFooBarStateTable [2] = {
73  { 1234 },
74  { 1234 }
75 };
76
77 int xxInputOutputTables [] = {
78  /* process Zero input table */
79  2,6,1,1,1,0,1,1,1,1,
80  /* process Zero output table */
81  2,6,1,1,1,0,1,1,1,1,
82  /* process One input table */
83  2,6,1,0,0,0,1,0,0,1,
84  /* process One output table */
85  2,6,1,0,0,0,1,0,0,1,
86  /* comma eater */
87  0
88 };
89
90 main (argc, argv)
91 int argc;
92 char **argv;
93 {
94  InterpretArguments (argc, argv);
95  (void) LaunchProcess (Envelope,
96     (int) FooBar,
97     (int) & (xxFooBarStateTable [0]),
98     sizeof (xxFooBarSTATE),
99     (int) xxFooBarInputs,
100    (int) xxFooBarOutputs,
101    (int) & (xxInputOutputTables [0]),
101 (int) &(xxInputOutputTables[10]),
102 0,
103 "Zero",
104 2048);
105 (void) LaunchProcess (Envelope,
106 (int) FooBar,
107 (int) &(xxFooBarStateTable[1]),
108 sizeof (xxFooBarSTATE),
109 (int) xxFooBarInputs,
110 (int) xxFooBarOutputs,
111 (int) &(xxInputOutputTables[20]),
112 (int) &(xxInputOutputTables[30]),
113 1,
114 "One",
115 2048);
116 (void) InitialProcedure ();
117 DispatchProcesses ();
118 (void) FinalProcedure ();
119 return (0);
120 }

Every model specification produces a procedure with the same name. In the example, the FooBar model produces the FooBar procedure (see line 22).

The input specification of a model produces a declaration of an array of strings. The name of the variable is xx followed by the model name followed by Inputs. In the example, the FooBar model’s input specification produces the xxFooBarInputs variable declaration (see line 10). The array is initialized with strings containing the input names.

The output specification of a model produces a declaration of an array of strings. The name of the variable is xx followed by the model name followed by Outputs. In the example, the FooBar model’s output specification produces the xxFooBarOutputs variable declaration (see line 16). The array is initialized with strings containing the output names.

The state specification of a model produces a structure type definition. The name of the type is xx followed by the model name followed by STATE. In the example, the FooBar model’s state specification produces the xxFooBarSTATE type definition (see line 6).

The initial state specification produces a structure initializer value. Its use is described below. In the example, the FooBar model’s initial state specification produces the values used on lines 72 and 73.

The body of the procedure generated from a model consists of a single switch (case) statement. Each action specification produces one of the cases in the case statement. The case selector is a bit mask in which the \(i^{th}\) bit is set if the \(i^{th}\) input is specified in the action selector. The number of an input is its index in the xx...Inputs array. For the initial action the \(n^{th}\) bit is set and for the final action the \((n+1)^{th}\) bit is set where \(n\) is the number of inputs. In the example where there are two inputs, action initial translates to case 4 \(2^2\), action final translates to case 8 \(2^3\), action none translates to case 0, action Beta translates to case 1 \(2^0\), action Alpha translates to case 2 \(2^1\), and action Alpha, Beta translates to case \(3 \left(2^0 + 2^1\right)\).

Inside the procedure references to the variables $time, $event, and $state are translated to references to the local variables xxTime, xxEventList, and xxState (see lines 55 and 59).

Each process specification produces a variable declaration. The declared variable is an element of an array. The name of the array is the name of the model of which the process is an instance preceded by xx and followed by StateTable. The type of the array is the type defined by the state specification of
the specified model. In the example, the process Zero and process One specifications produce the declaration of the two element array called xxFooBarStateTable (see line 71). The first entry in the array corresponds to process Zero, the second entry corresponds to processOne. The array is initialized to the value obtained from the initial state specification of the model. In the example, the xxFooBarStateTable array entries are initialized to 1234 (see lines 72 and 73).

Each process also has two integer sequences associated with it. These sequences are all stored in the common array called xxInputOutputTables (see line 76). In the example, the process Zero specification produces the entries on lines 78 and 80. The process One specification produces the entries on lines 82 and 84. These entries represent information derived from the connection specifications and the xx...Inputs and Wxx...Outputs declarations and the process on specifications. These arrays are interpreted as follows. The first number is the number of inputs or outputs that the process has. Let this number be \( n \). The first \( n \) numbers are indices into the array indicating where data about the \( i^{th} \) input or output \( 0 \leq i \leq n - 1 \) begins. Consider the entries on line 78 of the example. The first number in the array, 2, specifies that there are 2 inputs. Information about input number 0 begins at index 2 and information about input number 1 begins at index 6. (Note the double meaning of the first number.) Note that the name of input or output \( i \) can be obtained from the appropriate entry in xx..Inputs or xx..Outputs array entry. The information about an input or output consists of a number which specifies the fan-in or fan-out of the input. Let this number be \( f \). Following this number are \( f \) triples. The last number in each the xxInputOutputTables is just noise and should be ignored. Consider line 78 of the example. Information about input number zero begins at index 2 in the list. The number there is 1. This indicates that the fan-in is 1. Following this number, there is 1 triple, 1,1,0. The triples consist of a processor number, a process number, and an output number. This says that there is a process on processor 1, the number of that process is 1, and its output number 0 is connected to this input. The output table on line 80 is constructed in a similar fashion.

A single main procedure is emitted. This main procedure calls LaunchProcess once for each process specified in the source. A number of arguments are passed to the LaunchProcess procedure. Of note is the 9th argument, which is the process number assigned to that process. In the example, process Zero is assigned process number 0 on line 102.

Simulation begins when the main process calls DispatchProcesses. Before simulation begins, the initial procedure is called. After simulation ends, the final procedure is called. In the example, the initial procedure is InitialProcedure (line 116) and the final procedure is FinalProcedure (line 118).