

**THE YADDES  
DISTRIBUTED DISCRETE EVENT SIMULATION  
SPECIFICATION LANGUAGE  
AND  
EXECUTION ENVIRONMENTS**

*Bruno R. Preiss*

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

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

## INTRODUCTION AND MOTIVATION

- a single specification language for discrete event simulation (DES)
- independent of the underlying execution mechanism — sequential/parallel, conservative/optimistic
- change execution mechanisms without altering simulation models
- compare performance of different execution mechanisms
- target processor architectures: MIMD message-passing systems with no shared memory

## EXECUTION MECHANISMS

- sequential DES  
(event-list-driven)
- parallel DES mechanism using multiple, synchronized event lists (ultra-conservative)
- conservative parallel DES  
(without deadlock detection and recovery)
- optimistic parallel DES  
(using the virtual-time mechanism)

## MODELING METHOD

- real world system

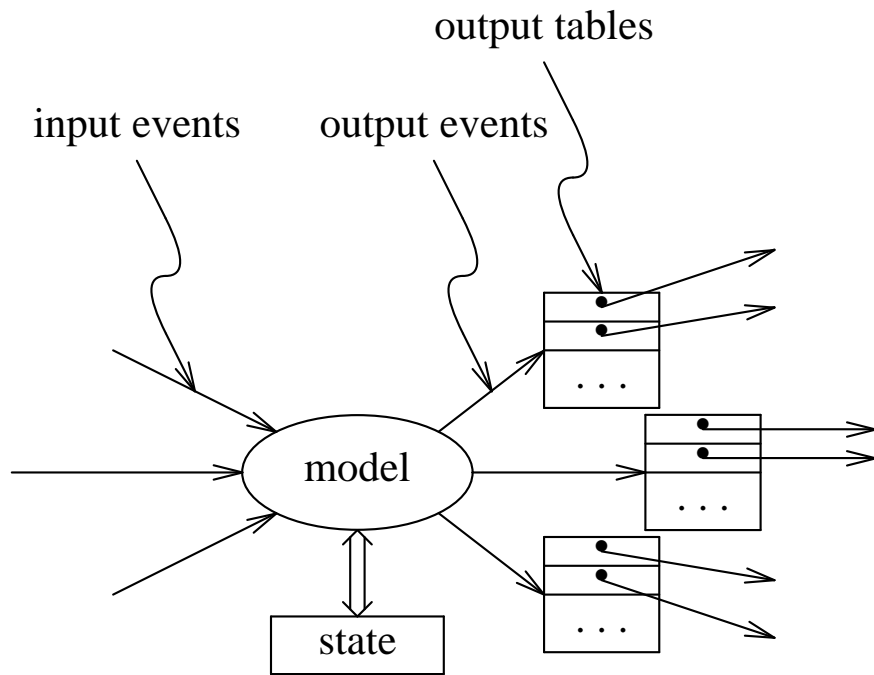
*modeled as:*

- a network of physical processes (PPs)
- static network topology

*simulated by:*

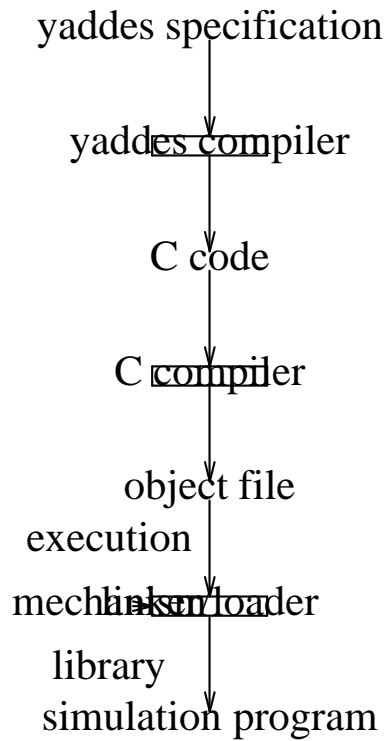
- static activation of logical processes (LPs)
- fixed IPC topology
- LPs are “general state machines”
- finite (albeit possibly very large) number of states
- specification of LP is called a *model*
  - state transition table *and*
  - output event specifications *for each*
  - input event combination

# A YADDES LOGICAL PROCESS



## THE YADDES LANGUAGE

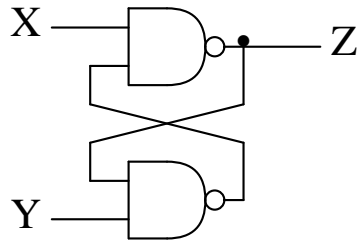
- preprocessor for the C programming language



## **LANGUAGE COMPONENTS**

- model specifications — describe generic LPs
- process specifications — instantiate LPs
- connection specifications — link LPs

## LOGIC CIRCUIT EXAMPLE: LATCH



## MODEL SPECIFICATION EXAMPLE

```
model TwoInputNand
  inputs in0, in1
  outputs out
  state { . . . }
  initial state { . . . }
  action initial { . . . }
  action in0 { . . . }
  action in1 { . . . }
  action in0, in1 { . . . }
end model
```

## MODEL EXAMPLE: TWO INPUT NAND GATE

```
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];
    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 EXAMPLE: READ EVENTS FROM INPUT FILE

```
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", $name);
      exit (1);
    }
    while (fscanf ($state->ifp, "%d%d",
      &time, &event) == 2) {
      OUTPUT ($out, time, event);
    }
  }
end model
```

## MODEL EXAMPLE: WRITE EVENTS TO OUTPUT FILE

```
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\n", $name);
      exit (1);
    }
  }
  action in {
    if ($event [$in] != $state->last) {
      $state->last = $event [$in];
      (void) fprintf ($state->ofp,
        "%d %d\n", $time, $event[$in]);
    }
  }
end model
```

## PROCESS SPECIFICATION EXAMPLE

```
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
```

## CONNECTION SPECIFICATION EXAMPLE

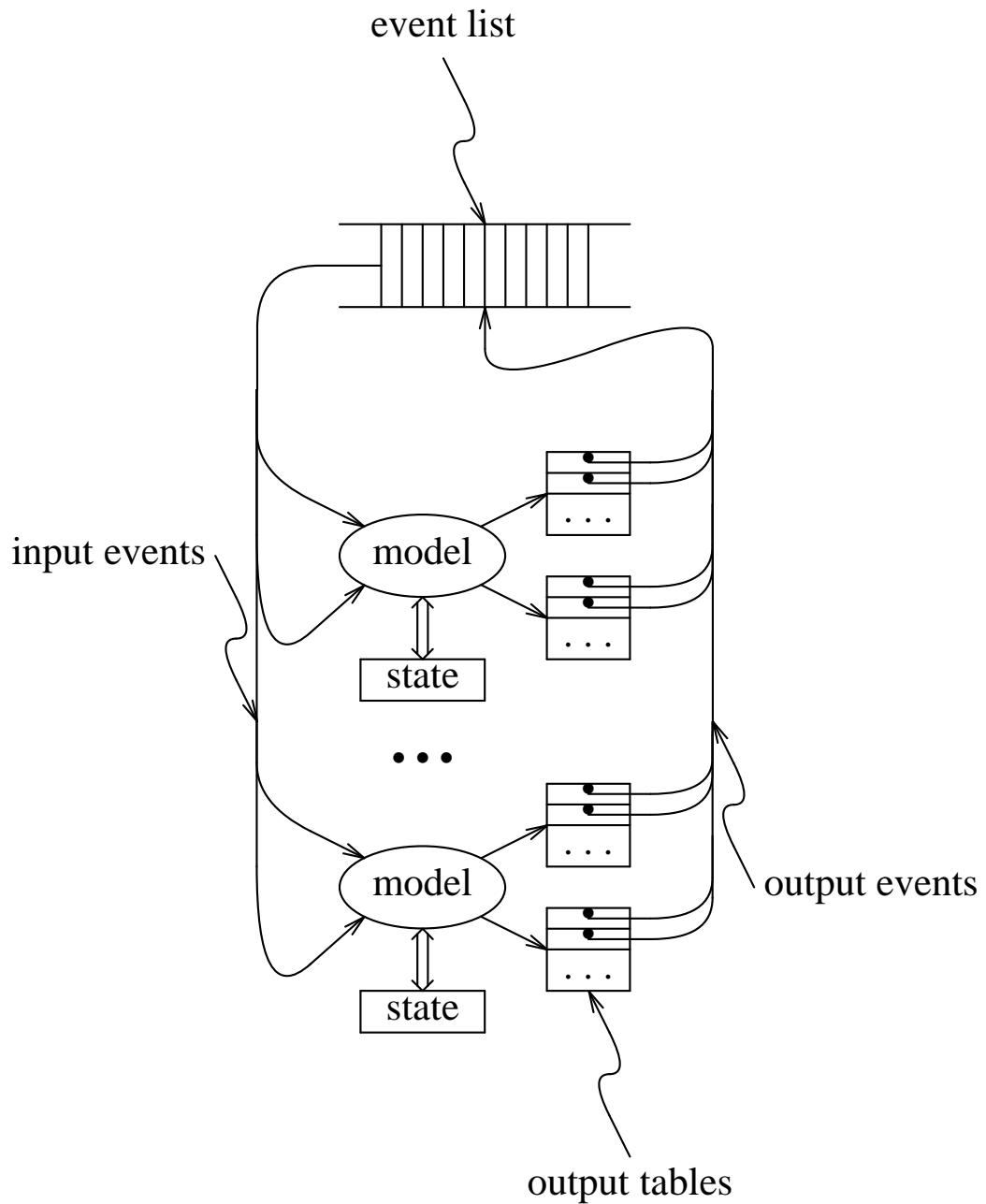
connect X.out to Gate0.in0

connect Y.out to Gate1.in1

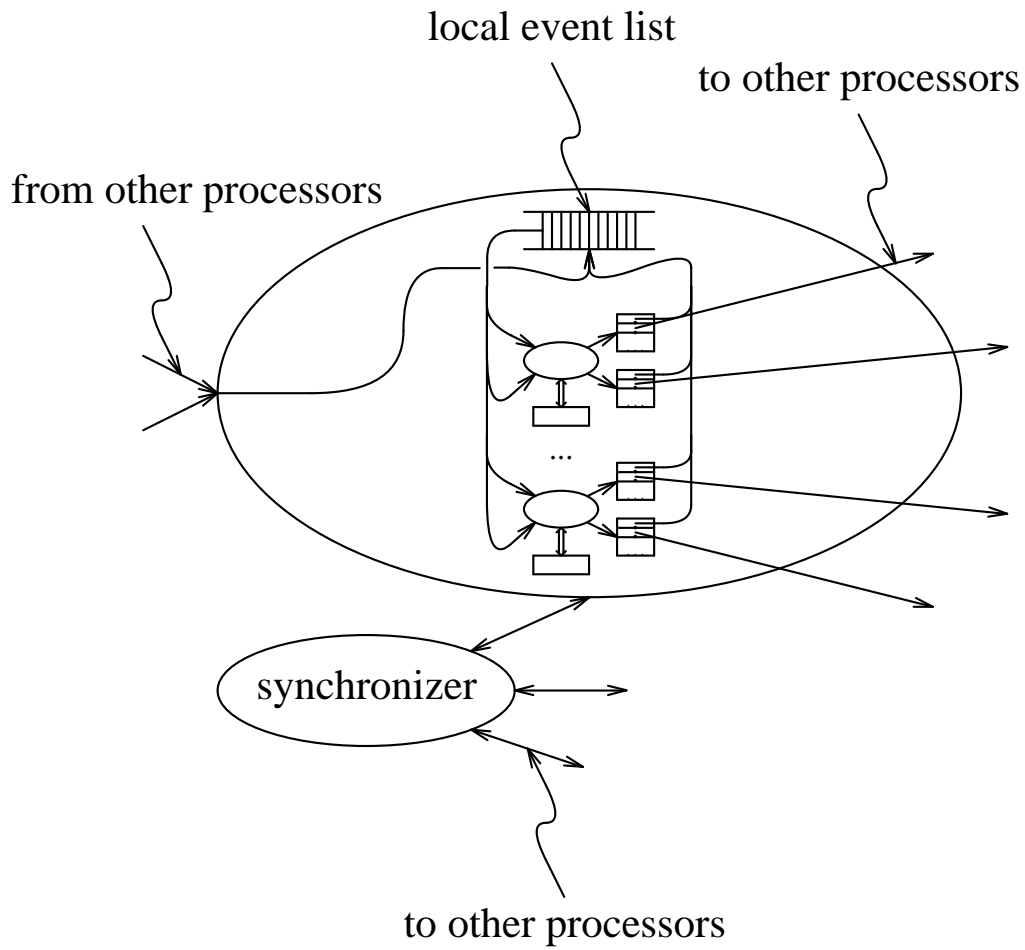
connect Gate0.out to Gate1.in0, Z.in

connect Gate1.out to Gate0.in1

# THE SEQUENTIAL SIMULATION ENVIRONMENT



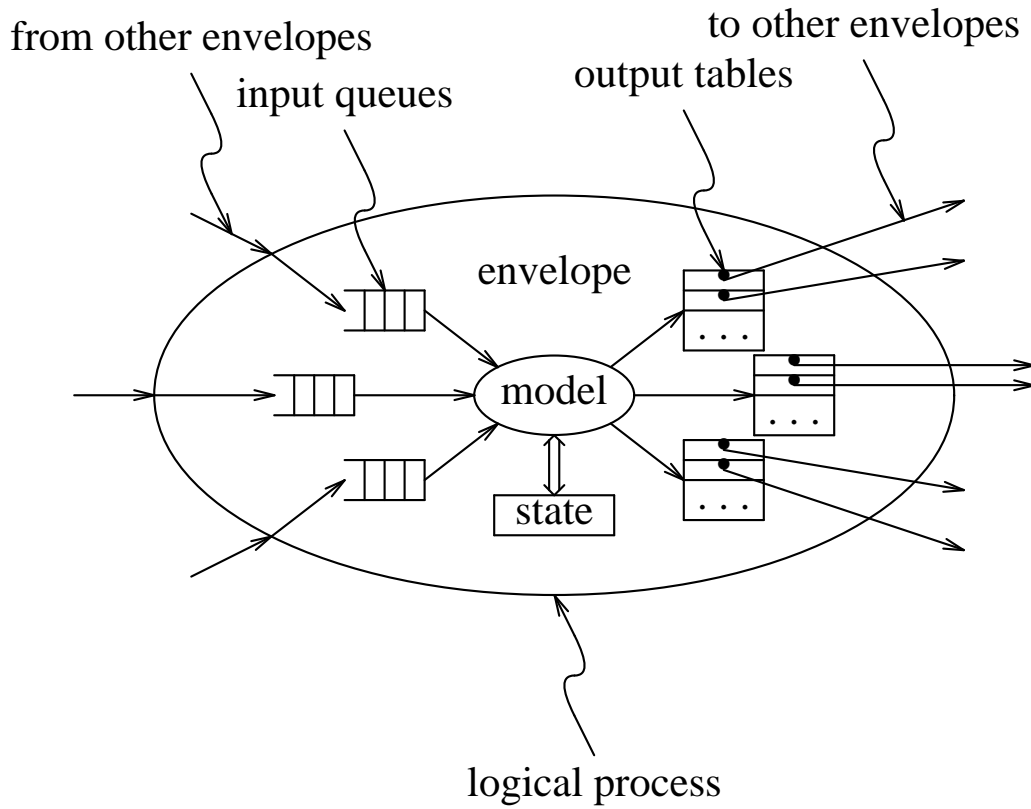
# PARALLEL DES USING MULTIPLE, SYNCHRONIZED EVENT LISTS



## **OPERATION OF MULTIPLE, SYNCHRONIZED EVENT LISTS**

- Basic cycle:
  - Each PE sends the time of its next event to the synchronizer.
  - The synchronizer selects the minimum and broadcasts it.
  - Each PE having the minimum removes events from its event list, forms event combinations, and invokes the appropriate logical processes' actions. Output events are either inserted event list or sent to another PE to be inserted into its event list.
  - When a PE is done, it sends a completion message to all its successors.
  - Each PE waits until it receives a completion message from all its predecessors.

# CONSERVATIVE PARALLEL DES



## **DEADLOCK AVOIDANCE IN CONSERVATIVE PARALLEL DES**

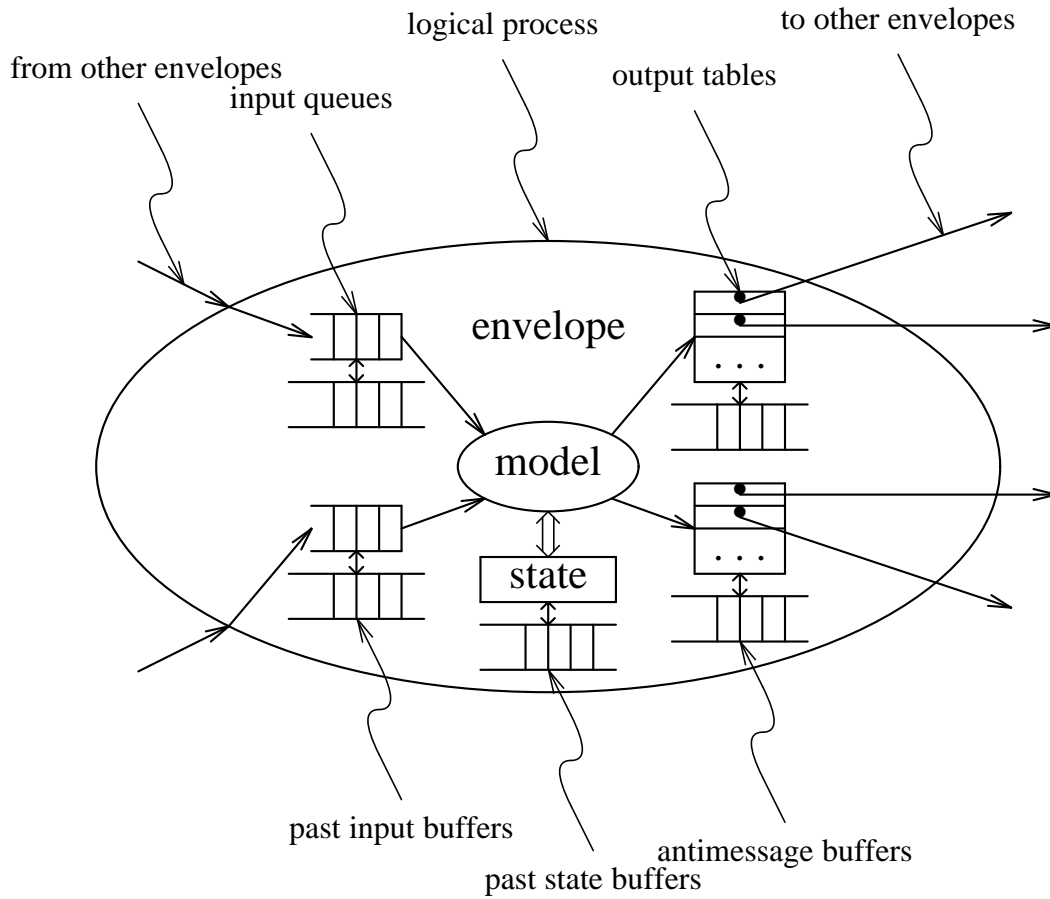
### **Null Messages:**

- Null messages are extra output messages 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.

### **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.

# OPTIMISTIC PARALLEL DES



## COMPUTING GLOBAL MIN TIME IN OPTIMISTIC PARALLEL DES

- Global min time is computed using a circulating token message.
- Each LP maintains two local variables:
  - $LT$  the current local time
  - $LT_{\min}$  the minimum value of local time since the last token message
- The token message contains three data items:
  - $i$ , the ID of the logical process that owns the token
  - $T_f$ , the previous value of global min time
  - $T_{\min}$ , a value used in the calculation of a new value for  $T_f$

## GLOBAL MIN TIME ALGORITHM FOR OPTIMISTIC PARALLEL DES

- Each LP executes the following algorithm upon receipt of the token:

receive  $(i, T_f, T_{\min})$  from  $LP_{(j+N-1) \bmod N}$

discard all information older than  $T_f$

if  $i=j$  then

$T_f \leftarrow LT_{\min}$

$T_{\min} \leftarrow LT$

else

if  $T_{\min} < LT_{\min}$

$T_{\min} \leftarrow LT_{\min}$

$i \leftarrow j$

end if

end if

$LT_{\min} \leftarrow LT$

send  $(i, T_f, T_{\min})$  to  $LP_{(j+1) \bmod N}$

## **OTHER FEATURES OF YADDES**

- multiple, independent, pseudo-random number streams
- statistics collection and reporting facilities
- execution statistics — message statistics, model call statistics, prediction statistics, overhead message statistics, roll-back statistics

## CURRENT PROJECT STATUS

- four execution mechanisms

three implementations:

- portable  
(uniprocessor version)

o/s:

- BSD 4.3 Unix
- Apollo DOMAIN/IX
- MS-DOS

processor:

- $\mu$ VAX II
- Apollo DN3010
- IBM-PC compatible
- VAX-only  
(simulated multiprocessor version)
- fully distributed  
(network of Apollo DN3010 workstations)

o/s: DOMAIN/IX with NCS and NIDL