Proceedings of the 1989 Winter Simulation Conference
E.A. MacNair, K.J. Musselman, P. Heidelberger (eds.)

A COMMON PROGRAMMING STRUCTURE FOR BRYANT-CHANDY-MISRA, TIME-WARP, AND SEQUENTIAL SIMULATORS

Marc Abrams
Distributed Systems Group
Department of Computer Science
Stanford University
Stanford, CA 94305-2140

Abstract

CPS expresses event-driven, parallel simulation programs. Rather than being a programming language, CPS is a structure imposed on an existing language, C++. CPS programs contain sufficient information to be implemented as Bryant-Chandy-Misra, time-warp, or sequential simulations. CPS is independent of any problem domain. We describe the main concepts of CPS, illustrating them using the problem of logic circuit simulation. We identify what information a parallel simulation must contain that is not required for sequential simulation.

1 INTRODUCTION

One major issue arises in writing parallel simulation programs: the simulation programmer (or *user*) must provide information not normally contained in a sequential simulation program. Furthermore, the information required varies, depending on which simulation protocol is used.

We describe a general-purpose programming system for parallel simulation called the Common Programming Structure. CPS is designed primarily for experimental comparison of the Bryant-Chandy-Misra (BCM) [3, 4], time-warp (TW) [9], and sequential simulation protocols. We believe that CPS may be extended to support simulation protocols proposed in the future.

CPS is unique in that the user writes a *single* program that mechanically generates three separate executable binaries, implementing each of the three simulation protocols listed above. CPS provides a common programming interface for all three techniques, giving rise to the "C" in CPS.

Mechanical generation of a sequential simulation is important to realistically estimate speedup of a parallel simulation protocol. This facility is also provided by the Time-Warp Operating System and Time-Warp Sequential Simulator [8].

One way to facilitate writing parallel simulation programs is to define a new programming language, as the Yaddes specification language [13] does. In contrast, CPS imposes a programming structure on an existing programming language (C++ [14]). This approach is also used in Sim++ [11].

CPS defines a library of C++ classes. A user creates a simulation program by writing C++ code that creates instances of CPS classes and calls members of the class instances created.

It is important to distinguish between a programming language and its implementation. This paper describes CPS, but not its implementation. One implementation of CPS is a system called OLPS [1].

We describe CPS by discussing how to write a logic gate simulator in sections 2 to 7. Then in section 8 we discuss some implications of using a single programming structure for multiple simulation protocols.

2 MODEL OF SIMULATION

The BCM and TW protocols use the following model of simulation [12]. The system to be simulated is viewed as a set of *physical processes* interacting via message passing. The simulation program is composed of a set of *logical processes* (LP's) which can predict the exact sequence of messages passed by the physical processes.

Example 1 *Throughout the paper we consider the example of simulating digital logic circuits. In this case each logic gate is a PP, and the changes in signal levels on wires interconnecting the logic gates are the messages.*

One could view a simulator as a directed graph in which each vertex corresponds to one LP, and the arcs correspond to possible paths over which messages may pass. A CPS program must specify two things for each LP:

- variables storing the current state of the LP
- sequential code that examines and updates state variables as well as generates new messages to be sent to other LP's

Example 2 *The specification of a logic gate by state variables and sequential code is an LP. Consider an LP representing a logic gate with N inputs. The output of the gate is the result of applying some function f to the inputs. One specifies the LP by:*

1. State variables: *$N + 2$ state variables are required:*
   - *One represents the number of input lines.*
   - *One represents the time delay that occurs before a change to an input produces a change to the output.*
   - *The remaining variables represent the last logic level (0 or 1) received on each of the N input lines.*
2. Sequential code: *The LP's sequential code is run each time a new message arrives. The code performs operation f on the logic level present in the message along with the state variables corresponding to the unchanged input arcs.*

3 TIME

We refer to the time units used during simulation as *simulation time*. Each LP in CPS has a state variable representing the current simulation time; this value is called the *local virtual time* (LVT), following Jefferson's concept of virtual time [10]. The *global virtual time* is a value less than or equal to the minimum of all local virtual times.

Whenever an LP sends a message to another LP, the sender includes a *time-stamp* with the message. The time-stamp is always the time at which the PP corresponding to the LP would have sent the message.

Example 3 *The time-stamp of a message sent from the LP for logic gate A to the LP for another gate B is the simulation time at which the output signal of gate A changes its value.*

4 MESSAGES

A simulation message in CPS is represented by an instance of class `Msg` (the typewriter font denotes fragments of CPS code), which is shown in Figure 1. In all figures presenting CPS classes, only the identifiers that the user normally needs to manipulate are shown. `SimTime`, `uschar`, and `usint` are typedef'd to `double`, `unsigned char`, and `unsigned int`, respectively.

```
class Msg {
public:
  SimTime Time;
  void*   Text;
  uschar  Priority;
  union   {
    usint      InArc;
    Sequencer* OutArc;
               //Sequencer explained later
  };

  Msg(SimTime time,          //constructor
    LP lp,
    void* text=IllegalAddr,
    usint SizeOfText=0;
    uschar priority=0);

  ~Msg();                    //destructor

  void SetSizeOfText(usint);
};
```

Figure 1: Class `Msg`

Figure 1 shows that class `Msg` has the following fields:

- `Time`: contains the time-stamp. In CPS, an object that contains a simulation time has the type `SimTime`.
- `Text`: contains a pointer (of type `void*`) to a user defined object, for example a logic level of 0 or 1 in the logic simulator. Class `Msg` functions as a container class, because its `Text` field can point to whatever object the user defines.
- `Priority`: only meaningful to classes `PrioritySmallest` and `PriorityLargest`, as described in section 5.2. (These classes implement a priority queueing mechanism.)
- `InArc`: identifies the source of the message received by an LP.
- `OutArc`: only used with class `MultiUserRoute` (see section 5.3).

Figure 1 also shows a constructor and a destructor for class `Msg`. The constructor shows that values for the

Text and Priority fields are optional. The Text field will default to the CPS defined constant IllegalAddr, indicating that the Text field does not contain a valid pointer. Constructor SizeOfText must be the length in units of char of the object pointed to by Text. (SizeOfText is ignored in a shared memory implementation of CPS.) Member SetSizeOfText is provided to allow change the length after the object is constructed.

Example 4 *Let L be an object of type* LP* *(i.e., a pointer to an object of type* LP*). L constructs a message to send to its downstream gates by calling the constructor of class* Msg. *As an example, suppose that L changes one of its outputs to logic level 1 at time 362.3. The message needed to contain this information is constructed and its address is stored in* M *as follows:*

```
char* NewLevel = new char;
NewLevel* = 1;
Msg* M = Msg(362.3, L, NewLevel, 1);
```

*Later on, the message may be destroyed by the statement "*delete M.*"*

When another LP receives message M, it can read the time-stamp of the message by accessing M->Time. In the logic simulator, the receiving LP can read the new logic level by accessing M->Text*.

5 LOGICAL PROCESSES

In CPS, the user must identify a *set of LP types* from which the simulator is constructed. The name of a type of LP must be a constant of type LPType. (LPType is typedef'd to long.) In a simulator there are usually multiple instances of each LP type.

Example 5 *For a logic simulator, there might be three types of LP's: two input AND gates, two input NOR gates, and INVERT gates. Usually the user defines a set of identifiers representing LP types by using the* enum *in C++:*

```
enum { AND, NOR, INVERT };
```

A logic simulator may simulate a circuit with 100 triple input AND gates, 59 dual input NOR gates, and 23 INVERT gates. The user assigns a constant of type LPId *(which is typedef'd to* long*) to uniquely refer to each LP instance. For example the NOR gates may be numbered 101, 102, ..., 159.*

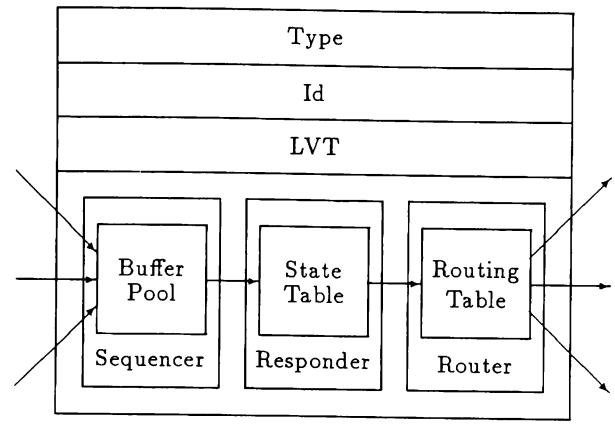

Figure 2: Components of an LP

The objects comprising class LP are shown in Figure 2. The components that contain data, but no member functions are:

Type: The value of this object distinguishes members of the set of LP types, as described above, for example AND from NOR gates.

Id: Any two instances of class LP that share the same value for Type must be assigned unique values for Id. Thus Id distinguishes NOR gate 101 from NOR gate 103.

LVT: This contains the local virtual time, which is set by the responder (see below).

The components that contain both data and member functions are:

Sequencer: Receives messages from other LP's into a *buffer pool.* Decides the order in which messages are presented to the responder. The buffer pool allows concurrent writing and reading by multiple LP's.

Responder: Simulates the PP that a given LP models. The responder contains the state variables described earlier.

Router: Sends messages generated by the LP to other LP's. The router may contain a routing table.

The C++ specification of class LP is shown in Figure 3. Note that class LP actually contains a pointer to the sequencer, responder, and router. Constructor argument TextEqual is discussed in section 5.4.

```
typedef Bool (*Equal)(Msg*,Msg*);

class LP {
public:
  LPType     Type;
  LPId       Id;
  SimTime    LVT;

  Sequencer* S;
  Responder* Re;
  Router*    Ro;

  LP(LPType type,
    LPId id,
    Equal* TextEqual=NULL);

  ~LP(void);
};
```

Figure 3: Class LP

5.1 How LP's Work

An LP repeats the following three actions:

1. The sequencer determines which message M in its buffer pool is the next one to be simulated. The sequencer may use LVT to make this decision. The sequencer removes M and returns it to the responder.

2. The responder simulates M and updates LVT and the state variables. During simulation the responder generates zero or more messages, and returns a list L of these messages to the router.

3. The router uses its routing table to send each message in L to downstream LP's by writing to the appropriate buffer pools.

Example 6 *Consider an LP in a logic simulation representing LP #127, whose type (from Example 5) is a two input NOR gate. The NOR gate LP repeats the following three actions (corresponding to the three actions above):*

1. *The sequencer examines all messages from either input arc that are waiting and chooses the one with the smallest time-stamp.* LVT *is not required to make this decision. (If multiple messages have identical time-stamps, the sequencer chooses one message arbitrarily.) The message chosen, M, represents the next logic level change that an input experiences. Let* M->Time*=362.3 and let the logic level be 1. Assume the message arrived on input #0.*

Sequencers:	*Routers:*
SingleSmallest	NullRoute
MultiSmallest	SingleRoute
PrioritySmallest	MultiUserRoute
SingleLargest	MultiProbRoute
MultiLargest	MultiCastRoute
PriorityLargest	BroadCastRoute

Table 1: Sequencers and responders available in CPS.

2. *Assume the last logic level received on input #1 was 0. Then the responder of the NOR gate would NOR together 1 and 0 (the values last received on inputs 0 and 1, respectively) and obtain 0. Further, let the delay in the NOR gate be 10 simulation time units. Then at LVT 362.3+10 the NOR gate begins to output logic level 0. (The NOR gate has now simulated up to time 372.3). The responder will change* LVT *and* M->Time *to 372.3 and* M->Text* *to (logic level) 0. (In this example,* LVT *need not be set, because neither the sequencer nor the responder of the NOR gate uses* LVT*.) The list returned to the router, L, will contain one message, M.*

3. *The router will place M in the buffer pool of the LP which is connected to the output of the NOR gate.*

CPS defines several sequencers and routers (Table 1). One may either use these or define his own. Responders are entirely problem dependent, and hence are always written by the user.

In C++ one normally creates class instances through the new operator. However, for any class listed in Table 1, which we denote C, the user calls a function whose name is the word *Create* concatenated to C, and whose arguments are the same as the constructor of C. This allows different sequencers and router implementations for the CMB, TW, and sequential protocols. (The user defines a compile time constant called Method, and sets its value to one of MethodBCM, MethodSeq, or MethodTW to specify which simulation protocol is to be used.)

The following subsections discuss sequencers, routers, and responders in detail.

5.2 Sequencers

Table 1 lists six sequencers. Sequencers whose names end in *Smallest* (*Largest*) return the message with the smallest (largest), time-stamp from the buffer pool. Sequencers whose names start with *Single* (*Multi*) allow single (multiple) input arcs to the LP.

```
class Sequencer {
public:
  LPId** InArcMap;
  virtual void PreLoad(usint NumCopies,
    Msg* M, uschar Priority=0);
};
```

Figure 4: Class Sequencer

A PrioritySmallest (PriorityLargest) sequencer provides a set of MultiSmallest (MultiLargest) sequencers, each having a unique priority level. Each message (see Figure 1) contains a Priority field, specifying on which MultiSmallest (MultiLargest) sequencer the message should be enqueued. PrioritySmallest (PriorityLargest) will select as the "next" message the smallest (largest) message from the highest priority, non-empty component sequencer.

Example 7 *In a simulator of a multi-processor computer, each processor may be considered a PP. Jobs waiting for execution at each processor may be messages. If jobs are scheduled by a first-in, first-out or a processor-sharing policy for each processor, a* SingleSmallest, MultiSmallest, *or* PrioritySmallest *sequencer may be used. If jobs are scheduled by a last-in, first-out policy, a* SingleLargest, MultiLargest, *or* PriorityLargest *sequencer may be used.*

All of the sequencers listed in Table 1 are derived from a class called Sequencer, whose declaration appears in Figure 4. Any user defined sequencers must also be derived from class Sequencer. Class Sequencer simply defines an interface common to all classes derived from it. Hence one creates instances of classes derived from class Sequencer.

Figure 4 contains a pointer to a table, InArcMap, and a function, PreLoad. InArcMap maps field InArc of an instance of class Msg to the Id of the LP that sent the message. Function PreLoad(N, M, P) constructs N copies of message M and inserts these messages into the buffer pool of the sequencer at priority level P (P=0 is the highest priority level). If there is only one priority level (as is the case in all classes but PrioritySmallest and PriorityLargest), P is ignored.

The declaration of the Create functions for sequencers is shown in Figure 5. Argument L specifies the LP that each sequencer is a component of. Sequencers PrioritySmallest and PriorityLargest have one sequencer for each priority level, which are identified by the numbers 0,1,...,HiLevel.

```
Sequencer* CreateSingleSmallest  (LP* L);
Sequencer* CreateMultiSmallest   (LP* L);
Sequencer* CreatePrioritySmallest(LP* L,
  usint HiLevel);
```

Figure 5: Declaration of functions that create sequencers (creation functions for *largest* sequencers is analogous)

```
class Router {
public:
  virtual Sequencer* AddRoute(LP*, Prob);
  virtual Sequencer* AddRoute(LP*);
  virtual Bool CheckProbRoute(void);
};
```

Figure 6: Class Router

5.3 Routers

Table 1 lists six routers. NullRoute destroys the messages it receives, as required by LP's acting as sinks. SingleRoute routes each output message to the same destination each time. MultiUserRoute uses field OutArc of each output message, set in the responder, to select an output route. MultiProbRoute selects an output route based on a set of probabilities. MultiCastRoute sends a copy of each output message to a set of LP's. BroadCastRoute sends a copy of each output message to all LP's.

All of the routers listed in Table 1 are derived from a class called Router, whose declaration appears in Figure 6. As is the case with class Sequencer, class Router defines an interface, and the user creates instances of classes derived from Router. In Figure 6, type Prob, representing a probability, is typedef'd to double, and type Bool is typedef'd to char.

Class Router has a member called AddRoute, which is used to specify the set of feasible output routes, as indicated below.

For SingleRoute: AddRoute(L), where L is of type LP*, will initialize the routing table to always select LP L as the output route.

For MultiProbRoute: AddRoute(L,P), where P is of type Prob, will initialize the routing table to select LP L as the output route with probability P. See CheckProbRoute, below.

For MultiUserRoute: `AddRoute(L)`, informs `MultiUserRoute` that `L` is a *potential* output route. (`MultiUserRoute` needs to know all the potential routes, because it may multicast simulation control messages to these LP's.) Furthermore, `AddRoute` also returns an identifier that the user later uses to select a particular output route in a responder. The identifier is of type `Sequencer*`. For example, an LP `L` may have two routes, `L1` and `L2`. The user calls `AddRoute` once for each output route:

```
Sequencer* S1 = AddRoute(L1);
Sequencer* S2 = AddRoute(L2);
```

After these calls, the user responder may route to `L1` or `L2` by setting field `OutArc` (see Figure 1) to `S1` or `S2`, respectively.

For MultiCastRoute: `AddRoute(L)` specifies that `L` is one of the LP's multicast destinations.

For NullRoute and BroadCastRoute: `AddRoute` is unnecessary and hence undefined.

Member `CheckProbRoute` is only called when using `MultiProbRoute`, and is called *after* all calls to `AddRoute` have been made. The member returns the constant `True` (which has a non-zero value) if and only if the sum of the type `Prob` field of all calls to `AddRoute` sum to one. This member should be called before simulation starts (via a call to `Net->Simulate()`, as described in section 6).

The Create functions for routers (e.g., `CreateNullRoute`, `CreateMultiProbRoute`) have one argument, of type `LP*`, which is a pointer to the LP that the router is a component of.

5.4 Duplicate Messages

It is possible for a sequence of two messages traveling over the same arc with non-decreasing time-stamps to be otherwise identical. We call the second message a *duplicate*. If the user gives argument `TextEqual` (see Figure 3) of the constructor of class `LP` a non-null value, then duplicate messages are to be detected and destroyed by sequencers and routers. (Duplicate destruction is called *event elision* by Davoren [5].) If the argument is `NULL`, then sequencers and routers will not spend time checking for duplicate messages.

Example 8 *In the logic gate simulator, assume that a NOR gate has last received a 1 on both inputs, so that its last output message had* `Text*` *= 0. The next input received over either input will generate a new output message with the same* `Text*` *value: 0. If the NOR gate LP was constructed with a non-null value for* `TextEqual`*, then the router will not send the second message, but rather will delete it. The reason is that the message does not indicate a change in signal level, and hence cannot affect the gate connected to the output of the NOR gate.*

```
class Responder {
public:
  SubState** CRWS;
             //Current read-write state

  Responder(LP* Owner, SubState**
    InitCRWS, usint NumSubStates,
    usint Frequency=0);

  void             Dirty(usint substate);
  virtual MsgList* RespondToUser(Msg* M);
  virtual Msg*     RespondToNull(Msg* M);
  virtual void     Commit(SimTime GVT){;}
};
```

Figure 7: Class `Responder`. `SubState` is typedef'd to `void`.

Sequencers and routers must determine when two messages are duplicates. To do this the user defined `Text` field of two messages must be compared. A user-written function is required for this. The address of the function is passed to the constructor for class `LP` as argument `TextEqual`. This function takes two type `Msg*` arguments, and returns either `True` or `False` if the `Text` fields are equal.

5.5 Responders

Responders are written by the user. However CPS provides a number of mechanisms to make writing responders easier. CPS defines a class called `Responder` (Figure 7), from which user responders are derived. As is the case with classes `Sequencer` and `Router`, class `Responder` defines an interface.

Recall from section 2 that an LP is specified by giving a set of state variables and a piece of sequential code. First we discuss the state variables, then we discuss the sequential code.

Certain aspects of the BCM and TW protocols become visible to the user at this point in the sense that the design of the responder affects the efficiency of the simulator. The user must structure the state variables needed by a responder differently for the BCM and TW protocols.

State variables. The state of an LP must be partitioned by the user into two categories, read-only and read-write states.

Read-only state: The user defines a class that contains all read-only state variables.

Read-write state: The user may partition the read-write state into a set of *sub-states*. The number of sub-states is specified as argument `NumSubStates` in the constructor for class `Responder`. The argument is optional; its absence indicates that partitioning is not desired.

Sub-states are defined to improve the efficiency of TW simulation. The responder simulates in response to a message removed by the sequencer from its buffer pool. The TW protocol requires that multiple versions be kept of the LP's read-write state variables. Old versions are used when the LP must roll back its local virtual time.

Creation of a new version of the state requires allocating memory and copying the current state content. The concept of sub-state is provided by CPS to reduce the overhead of creating new versions. The TW implementation manages versions on a per sub-state basis. Therefore the user can partition the read-write state into a sufficiently small number of sub-states so that a new version of only the state component actually changed during simulation of a message will be created. However the user should not create too many sub-states, because this increases the space overhead required to manage versions. Sub-state partitioning is irrelevant when the BCM protocol is used.

The sub-states that the user defines are numbered 0,1,...,`NumSubStates-1`. (Usually the user will define constants using `enum` with which he refers to the sub-states.) The size of the array is `NumSubStates`. Responder pointer `CRWS` contains the current read-write state of the LP. The TW implementation will determine the correct current local state before activating user-written code. This may involve saving a new state or restoring an old state and initiating rollback at downstream nodes. The BCM protocol will not modify `CRWS`.

The `Frequency` argument for the class `Responder` constructor specifies how often to create a new version for a changed object. A value of q will not generate a new version of a sub-state for the first q calls to `RespondToUser` that do not write to the sub-state, while a new version is created on call number $q+1$.

Example 9 *If one were writing a logic simulator, he would need to write a responder for each type of logic gate. From Example 2, the logic gate simulator for an N input logic gate requires $N+2$ state variables:* `NumInputs`, `Delay`, *and a size N array called* `LastInput`.

```
struct RO_State {
  usint NumInputs;
  SimTime Delay;
  RO_State(usint numinputs, SimTime delay)
    { NumInputs = numinputs;
      Delay = delay; }
};

struct RW_SubState {
  Bool LastInput;
  RW_SubState(Bool lastinput=False)
    { LastInput = lastinput; }
};
```

Figure 8: State variables necessary for logic simulator responder.

The values of `NumInputs` *and* `Delay` *are never changed; hence the read-only state consists of these two variables.*

One way to partition the remaining (read-write) state variables is to make each variable `LastInput` *a sub-state. Consider a two input logic gate, in which input #2 is tied to ground. In this case one state variable will never change its value; hence only one version of the variable need ever be created. Meanwhile with the other variable in its own sub-state, the operations of allocating a new state and copying the old value to it are faster than if sub-states were not used. Other examples occur when one of the two inputs is used less frequently than the other, or when signals arrive on the two inputs in non-overlapping periods of time.*

The necessary declarations of state variables appears in Figure 8.

Sequential code. The user provides members `RespondToUser` and `RespondToNull`.

`RespondToUser` is the heart of a simulation program, because it contains the actual algorithm to simulate the behavior of a PP. This member is called with the next (non-null, or user) message returned by the sequencer to be simulated as its argument. `RespondToUser` then updates the local state, `CRWS`, and returns a list of zero or more messages to be sent to downstream LP's.

The user must call member `Dirty` one time for each sub-state modified. Furthermore `Dirty` must be called *before* the first modification occurs, to insure that a running LP can be preempted. `Dirty` takes one argument, specifying the identity of the sub-state that was modified. Calls to `Dirty` are necessary so that the TW implementation can properly manage sub-state versions. Calls to `Dirty` have not effect in the BCM implementation.

```
typedef MsgList* (*LGF)(Msg*, usint,
                        SubState*);

struct LG : Responder {
  RO_State* ROS;
  LGF f;

  LG(LP* Owner,
    LGF f,
    usint numinputs,
    RO_State RO_InitState,
    RW_SubState RW_InitSubState) :
    ( Owner, RW_InitSubState, numinputs)
    { F = f;    ROS = RO_InitState; }

  MsgList* RespondToUser(Msg* M)
    {
    LVT = M->Time;
    Dirty(M->InArc);
    ( (RW_State*) CRWS[M->InArc] ) ->
      LastInput = *M->Text;
    return (*LGF)(M,ROS->NumInputs,CRWS);
    }

  Msg* RespondToNull(Msg* M)
    {
    M->Time += ROS->Delay;
    return M;
    }
};
```

Figure 9: User-written responder for any gate in the logic simulation example.

Member RespondToNull is only called by the BCM protocol. Its argument, M, is a null message sent to the LP. The LP must generate a single null message and multicast it over all output routes. Typically RespondToNull calculates from LVT a lower bound on when the LP will next send a non-null message, and stores this value in M->Time, and returns M.

Member Commit is optionally written by the user. It is called with increasing values of GVT as an argument to allow the user to commit any actions that he defines which can be rolled back. An example would be user input or output operations involving istream or ostream.

Example 10 *In the logic gate simulator, it is possible to define one class that represents any type of gate (e.g., AND, OR, NOR, etc.). This class,* LG, *is shown in Figure 9.*

Class LG *contains two objects:* ROS *and* f. ROS *is set by the constructor of class* LG *to point to the read-only state.*

```
MsgList* NOR(Msg* M, usint NumInputs,
             SubStateCRWS)
{
  Bool rv = False;
  for (usint i=0; i<NumInputs; i++)
    rv |=
      ((RW_State*) CRWS[i])->LastInput;
  return !rv;
}
```

Figure 10: An example of a type LGF function.

Object f *is set by the constructor to point to a function that actually does the AND, OR, NOR, etc., operation of the gate. One such function is given in Figure 10.*

Note that because class LG *is derived from class* Responder, *the constructor of class* Responder *is called when the constructor of class* LG *is called. The class* Responder *constructor is responsible for constructing the initial version of the read-write state, using* RW_InitState *as its initial value.*

During simulation, member RespondToUser *is called each time a message is routed to a gate. When* RespondToUser *returns, the router associated with the LP to route the resultant message. Member* RespondToUser *performs the following actions:*

- *Call member* Dirty *to indicate that one sub-state has been modified.*
- *Use the* InArc *field of* M *to identify which input level changed. The logic value for this input (in* *M->Text*) is stored in the read-write state.*
- *Update the time-stamp of* M *by incrementing it by* Delay.
- *Call the function pointed to by* LGF, *which might be the NOR function of Figure 10. Function NOR uses the C++ logical or operator to combine all read-write state variables to calculate a new output for the gate.*

A logic gate is guaranteed not to change its output until Delay *units after the time-stamp contained in the null message. Hence member* RespondToNull *increments the time-stamp of a null message by* Delay.

Example 11 *Shown below is a fragment of C++ code that a user might write to create an LP for a two input*

NOR gate with a delay of 11.3 time units in a logic simulator. One first constructs the LP; then he constructs a sequencer, responder, and router; and finally he sets the LP to point to the sequencer, responder, and router:

```
LP* L = new LP(NOR, 159);
L->S  = CreateMultiSmallest(L);
L->Re = new LG(L, &NOR, 2,
            new RO_State(2, 11.3),
            new RW_SubState[2]);
L->Ro = CreateSingleRoute(L);
```

This type of code fragment is usually used before simulation starts, when parsing a file containing a description of the model to be simulated.

6 INITIATING AND TERMINATING SIMULATION

A complete simulation program starts by reading a file containing the model description, allocating the necessary LP's, and setting the routing via `AddRoute`. The program then simply executes "`Net->Simulate()`" to run the simulation. Then the user terminates the simulation program itself by executing "`Net->Terminate()`."

7 INPUT AND OUTPUT

C++ provides classes `istream` and `ostream` for input and output operations. The user may only operate on these streams before calling `Net-> Simulate()` (e.g., to read a set of parameters whose value does not change after simulation starts) or after `Net->Simulate()` returns (e.g., to dump data collected in memory to disk).

During simulation (i.e., during execution of `Net-> Simulate()`), the user must operate on classes `pistream` and `postream`, which are defined by CPS. The names of these classes are chosen to emphasize that an instance of either class is *private to a single LP.*

Class `pistream` adds the following semantics to C++ class `istream`. An instance of class `pistream` is associated with an instance of class `streambuf`. (Class `streambuf` buffers values read from a file.) When operator >> (i.e., the read operator) is called by an LP with LVT l, the value returned by >> is such that all preceding values in the `streambuf` were returned to the LP when the LP had an LVT that did not exceed l.

An instance of class `postream` is associated with an instance of class `streambuf`. (Class `streambuf` buffers values read from a file.) When operator << (i.e., the write operator) is called by an LP with LVT l, the argument given to << will not be added to the `streambuf` until the GVT is greater or equal to l.

One other form of output is provided: class `Histogram`. This class is also private to a single LP. The class implements a histogram that is stored in memory during simulation, and then is written to a file during `Net->Terminate()`.

8 IMPLICATIONS OF A COMMON PROGRAMMING STRUCTURE

We started this paper by observing that writing a parallel simulation differs from writing a sequential simulation in that additional information has to be provided by the programmer. It is instructive to identify what this information is for the BCM and TW protocols.

BCM protocol: Responder member `RespondToNull` is mandatory if the protocol is implemented using deadlock avoidance.

TW protocol: Providing the information listed below as optional may improve the efficiency of the simulation.

- (Mandatory) References to state variables for each LP have to be made through pointer variables. Specifically, pointer `CRWS` is needed. (The pointer is necessary because its value, representing the current state, is manipulated by the TW implementation invisibly to user written code.) The BCM protocol requires such pointers.
- (Optional) Responder member `Dirty` is required for the TW implementation to decide when to allocate a new version of a state variable.
- (Optional) The user must choose an efficient partition of the state variables in each LP.
- (Mandatory) LVT must always be updated. In contrast LVT need only be updated for the BCM protocol when sequencers other than `SingleSmallest`, `MultiSmallest`, and `PrioritySmallest` are used. The logic gate example requires no updates to LVT for the BCM protocol.
- (Optional) The user must decide on how frequently new versions should be created.

We note that if the memory management system supports versions (for example Fujimoto's Virtual Time Machine [7]), then the user would not have to refer to states through pointer variables and would not have to call member `Dirty`.

Both BCM and TW: A function of type Equal must optionally be provided if duplicate messages are to be detected. Furthermore, the user must use remember to use I/O streams that are private to an LP (i.e., pistream and postream) during simulation. Moreover, the user should minimize the amount of input and output operations required during parallel simulation. For example, multiple processes frequently reading from a a pistream that uses a single disk as the backing store may substantially increase the simulation running time.

CPS does not currently address on important issue in writing parallel simulation programs: look-ahead. Look-ahead is recognized to be crucial to good performance both in the BCM [6] and the TW protocols [2]. When using CPS, the user must implement look-ahead in an *ad hoc* manner. More understanding of look-ahead is required before a look-ahead mechanism can be incorporated into CPS.

One comment may be made about the logic simulator example. CPS defines an interface that is independent of the problem-domain. In the logic simulator, a second interface (function LGF) is defined; many implementations of LGF are then provided to implement the specific logic gate types (i.e., AND or INVERT gates). Interface LGF is specific to the problem domain. The lesson is that for a given problem, one should try to develop an interface like LGF. In this manner the simulator can represent an enormous class of problems with a small amount of code. This is of great benefit in experimental work that seeks to characterize which simulation protocols work best in which problem domains.

9 ACKNOWLEDGEMENTS

This work was sponsored in part by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211, as well as equipment from Digital Equipment Corporation. The use of sub-states in section 5.5 was influenced by Richard Fujimoto's Virtual Time Machine [7].

References

[1] M. Abrams. The Object Library for Parallel Simulation (OLPS). *Proc. Winter Simulation Conference*, San Diego (Dec. 1988) 210-219.

[2] D. Baezner, *et al.* Algorithmic Optimizations of Simulations on Time Warp. *Dist. Sim. 1989.* Soc. for Comp. Sim., Tampa (March 89), 73-78.

[3] R. E. Bryant. *Simulation of Packet Communication Architecture Computer Systems.* Tech Rep. MIT,LCS,TR-188, M.I.T., Cambridge, MA (1977).

[4] K. M. Chandy, V. Holmes, and J. Misra. Distributed Simulation of Networks. *Computer Networks 3,* (1979) 105-113.

[5] M. Davoren. A Structural Mapping for Parallel Digital Logic Simulation. *Dist. Sim. 1989.* Soc. for Comp. Sim., Tampa (March 89), 179-182.

[6] R. M. Fujimoto. Lookahead in Parallel Discrete Event Simulation. *Proc. 1988 Int. Conf. on Parallel Processing* (Aug. 88) 34-41.

[7] R. M. Fujimoto. Virtual Time Machine. *Proc. ACM Symp. on Parallel Alg. and Arch.* (June 89).

[8] P. Hontalas, *et al.* Performance of the Colliding Pucks Simulation on the Time Warp Operating Systems (Part 1: Asynchronous Behavior and Sectoring). *Dist. Sim. 1989.* Soc. for Comp. Sim., Tampa (March 89), 3-7.

[9] D. Jefferson and H. Sowizral. Fast Concurrent Simulation Using the Time Warp Mechanism. *Dist. Sim. 1985.* Soc. for Comp. Sim., San Diego (Jan. 85), 63-69.

[10] D. R. Jefferson, Virtual Time. *ACM Trans. Prog. Lang. Syst. 7,* 3 (July 1985), 404-425.

[11] G. Lomow. *Sim++.* Jade Simulations Inter. Corp.

[12] J. Misra. Distributed Discrete-event Simulation. *ACM Computing Surveys 18,* 1 (March 1986), 39-66.

[13] B. R. Preiss. The Yaddes Distributed Discrete Event Simulation Specification Language and Execution Environments. *Dist. Sim. 1989.* Soc. for Comp. Sim., Tampa (March 89), 139-144.

[14] B. Stroustrup. *The C++ Programming Language.* Addison Wesley, Reading, MA (1986).

AUTHOR'S BIBLIOGRAPHY

MARC ABRAMS received a Ph.D. in Computer Science from the University of Maryland in 1986. He has been a post-doctoral scholar at IBM and Stanford University. He developed simulation models for the U.S. Army. He is currently a Visiting Assistant Professor of Computer Science at Virginia Tech, and consults for Jade Simulations International Corporation. His research focuses on the performance of parallel software.

Marc Abrams
Dept. of Computer Science
Virginia Tech
Blacksburg, VA 24061-0106
(703) 231-6931
marc@vtodie.cs.vt.edu