*Proceedings of the 1995 Winter Simulation Conference*
*ed. C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman*

# COMPOSE: AN OBJECT-ORIENTED ENVIRONMENT FOR PARALLEL DISCRETE-EVENT SIMULATIONS

Jay M. Martin
Rajive L. Bagrodia

Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90024

## ABSTRACT

Existing environments for parallel discrete-event simulation provide support for either conservative or optimistic algorithms, with very few supporting both. This paper describes a parallel simulation environment that supports the execution of a model using an existing adaptive simulation algorithm where submodels may be synchronized using conservative or optimistic algorithms, and an object may dynamically change its mode of synchronization. The environment has been designed as a C++ class library and has been implemented on an IBM SP2 multicomputer.

## 1 INTRODUCTION

Parallel simulation languages (PSLs) and libraries provide programmers with a set of model definition primitives together with a set of parallel programming primitives for process (or thread) definition, creation, and interprocess communication and synchronization. Researchers have provided these constructs either as language extensions or as functions implemented as library routines. The former approach has the advantage that the compiler or preprocessor can provide strict type checking and sophisticated code generation and optimization capabilities. However, it has the distinct disadvantage of requiring the programmer to learn new constructs and perhaps an entirely new set of program development tools. The primary advantage of a library-based approach is that the programmer can continue to use a familiar programming environment.

COMPOSE (Conservative, Optimistic and Mixed Parallel Object-oriented Simulation Environment) is a new object-oriented environment for parallel discrete-event simulations. A number of languages and libraries for parallel simulation have been defined including YADDES (Preiss 1989), OLPS (Abrams 1988), SPEEDES (Steinman 1991), Sim++ (Baezner, et al. 1990), Maisie (Bagrodia and Liao 1994), MOOSE (Waldorf and Bagrodia 1994), ModsimII (Bryan 1989) and SCE (Gill, et al. 1989). Almost all preceding environments support either conservative or optimistic approaches, with very few supporting both. To the best of our knowledge, no existing simulation environment allows a model to be executed using an adaptive protocol, where individual simulation objects in the model may dynamically change their execution mode from conservative to optimistic and vice-versa.

The environment is designed around the C++ language and the simulation facilities are provided as library routines rather than language enhancements. This allows a C++ programmer to use the simulation system without learning new constructs. The programmer may also use an existing C++ program development environment to design parallel simulations. The library-based simulator has been implemented on a 128-node IBM SP2 distributed memory parallel computer. The implementation can use either conservative (Misra 1986) or optimistic synchronization (Jefferson 1985; Jefferson and Sowizral 1985) algorithms, and may also use a mixed synchronization scheme where different subsystems of the model are synchronized using conservative or optimistic mechanisms (Jha and Bagrodia 1994).

A parallel simulation environment must provide parallel programming facilities that include process (or object) creation, and interprocess communication and synchronization, together with event scheduling constructs. The computation model used by COMPOSE assumes that a parallel program is composed of a set of processes that do not share state and communicate exclusively by asynchronous message passing (i.e., we conservatively assume a distributed memory model). The class library provides routines for (remote) process creation, termination, communication, and synchronization. Methods are also provided to schedule conditional and unconditional events.

## 2 CLASS LIBRARY FRAMEWORK

**Simulation Entities** It is natural to represent an object in the simulation (henceforth referred to as an "entity") as a C++ object which is an instance of some C++ class. For this purpose, the COMPOSE library provides a base entity class called *BaseEntityType* which provides a set of useful member functions (operations) and encapsulates information necessary for the simulation environment. All entities in the simulation must be derived from this *BaseEntityType*. The following is an outline of this class:

```
class BaseEntityType {
 protected:
   // Simulation operations used by deriving entities.
   // Such as getting and advancing simulation time,
   // sending messages, etc.
   SimTimeType CurrentTime() const;
   void Hold(SimTimeType);
   void SendMessage(...);
   void IAmOptimistic();
   ...
 private:
   ... // Environment implementation details.
};
```

**Messages** COMPOSE uses messages as the primary communication mechanism among entities. Messages are implemented as C++ objects, where each message type is derived from a base class called *BaseMessageType*:

```
class BaseMessageType {
   TimeType Timestamp;
   EntityIDType Sender;
   EntityIDType Receiver;
   // And other COMPOSE system message parameters
   // common to all messages.
};
```

Message types used in the program are derived from the base type as illustrated by the following example:

```
class UserMessageType: public BaseMessageType {
   public:
     // User-specified parameters.
     int  Parm1;
     char Parm2;
};
```

COMPOSE messages must be self-contained and thus must not contain pointers.

**Sending Messages** To send messages among entities, the *BaseEntityType* provides a method called *SendMessage*:

```
void SendMessage(const EntityIDType& EntityID,
                 const BaseMessageType& AMessage);
```

Method *SendMessage* gives the message to the underlying communication system which computes the location of the receiver process (by looking at its *EntityID*) and transmits the message to the destination.

The *SendMessage* method is overloaded to provide an alternate implementation which instead accepts a pointer to the message so that extra copying can be avoided. This pointer is set to *NULL* to prevent the sent message from being modified.

```
void SendMessage(const EntityIDType& EntityID,
                 BaseMessageType*& AMessagePtr);
```

Each message type can provide a constructor to allow for easy initialization of message parameters prior to transmission. For example:

```
SendMessage(EntityID,
            new UserMessageType(Parm1,Parm2...));
```

**Message Processing** COMPOSE uses the concurrent object model for message processing. When a message is received, a method of the object is executed to completion. This is in contrast to the process model which has explicit receive statements and thus, the program-counter is part of its state. The code to be executed by an object on receipt of a message is specified as a method of that object with a parameter matching the type of the message. A runtime binding mechanism is needed to associate a message type with its corresponding method. This binding framework, which takes control of method invocation, is necessary because of the need to schedule events in simulation order and because the simulation objects are distributed. These capabilities could be transparently provided in a concurrent language by a compiler, but this would conflict with COMPOSE's main design goal of a completely library-based implementation with no special compilers or preprocessors.

The library provides a routine *BindMethod* for runtime message to method binding which is usually called in the object's constructor:

```
BindMethod(UserEntityType, JobMethod, JobMessageType);
```

Failure to properly bind a method or sending a message to an entity that it does not recognize will result in a runtime error message. Note that this makes COMPOSE's message passing mechanism dynamically typed.

**Entity Creation and Initialization** To create an entity, we simply execute the following statement which creates an entity on a certain node and returns an entity ID:

```
CreateEntity(UserEntityType, WhichNode, EntityID);
```

After creation, the entity must be initialized by sending an initialization message. The entity indicates the initialization message type and method to the runtime by executing a variant of the *BindMethod* routine called *BindInitMethod*.

**Example Entity Class Definition** To illustrate these features we show a class definition for a simple FIFO server:

```
class ServerEntityType: public BaseEntityType {
 public:
   COMPOSE_EntitySetup(ServerEntityType);
 private:
   // Define entity methods.
   void InitMethod(const InitMessageType& InitMessage);
   void JobMethod(const JobMessageType& JobMessage);

   // Define constructor with BindMethod statements.
   ServerEntityType() {
     BindInitMethod(ServerEntityType, InitMethod,
                    InitMessageType);
     BindMethod(ServerEntityType, JobMethod,
                JobMessageType);
   }

   // Declare state variables.
   ...
};//ServerEntityType//
```

The structure of a COMPOSE entity is the same as that of a C++ class, with *public* and *private* data and member functions. In this class definition two methods, *InitMethod* and *JobMethod*, are declared with message parameters of type *ServerInitMessageType* and *JobMessageType*, respectively (these types can be defined elsewhere or can be nested in this class). The constructor *ServerEntityType*() is defined and contains two bind statements which bind the two methods to their message types. The state variables are declared at the end of the class (omitted for brevity). The *COMPOSE_EntitySetup* macro inserts declarations into the class needed by COMPOSE for entity creation.

Given a set of entity class definitions, the programmer can build a static topology simulation by first creating the entities using *CreateEntity* in a simulation "driver" routine. Topology information is then passed to these entities in the initialization messages. The driver can then issue events to start the simulation.

**More Advanced Features** To this simple base, COMPOSE adds boolean guard methods which can be used by an object to implement selective receives. Each bound method can be specified with an optional guard function using the *BindMethodWithGuard* operation instead of *BindMethod*. The COMPOSE runtime will automatically buffer messages for an entity if the corresponding guard function for that message returns FALSE. COMPOSE also provides methods to provide conditional timeout events. A timeout event is one that is executed only if another event does not occur during the timeout duration. Timeouts can be used at any point during the execution of the simulation and are called from an entity's methods as follows:

```
SetupTimeout(TimeoutMethod, TimeoutTime);
```

This code indicates that the entity will execute the *TimeoutMethod* at *TimeoutTime* unless it is interrupted by an intervening event.

## 3 SYNCHRONIZATION PROTOCOLS

In a parallel discrete-event simulation, each simulation process or entity must eventually process incoming messages in their global timestamp order. Enforcing this requirement, referred to as the causality constraint, is the central problem in efficient execution of parallel simulations. Two primary approaches have been suggested to solve the synchronization problem: conservative and optimistic.

Conservative algorithms do not permit any causality error: each object in the simulation processes an incoming message only when the underlying synchronization algorithm can guarantee that it will not subsequently receive a message with a smaller timestamp. This constraint may introduce deadlocks, which are typically avoided by using 'null messages.' A null message is a timestamped signal sent by a LP to indicate to other LPs a lower bound on the timestamp of its future messages. In general, the greater this interval, the better its performance with conservative protocols. Efficient implementation of null messages is also facilitated if each LP maintains

the set of its source and/or destination LPs. (Otherwise, *null* messages may be needlessly broadcast to all LPs).

In optimistic protocols, a LP is allowed to process events in any order; however, the underlying synchronization protocol must detect and correct violations of the causality constraint. The simplest mechanism for this is to have each LP periodically save (or checkpoint) its state. Subsequently, if it is discovered that the LP processed messages in an incorrect order, it can be rolled back to an appropriate checkpointed state, following which the events are processed in their correct order. The rollback may also require that the LP unsend or cancel the messages that it had itself sent to other LPs in the system. An optimistic algorithm is also required to periodically compute a lower bound on the timestamp of the earliest global event, also called the Global Virtual Time or GVT. As the model is guaranteed to not contain any events with a timestamp smaller than GVT, all checkpoints timestamped earlier than GVT can be reclaimed. Thus, the primary facilities needed to implement optimistic methods include *checkpointing*, *message cancellation*, *rollback and recomputation*, and *GVT computation*.

Recently, a new protocol has been suggested that allows each LP to individually select either the conservative or the optimistic execution mode (Jha and Bagrodia 1994). This protocol defines a local metric called EIT (for Earliest Input Time) for each LP. Conservative LPs can process all events that are timestamped earlier than its EIT, whereas optimistic LPs can use EIT to reclaim memory. A separate global control mechanism is defined to allow each LP to periodically update its EIT. The global control mechanisms could use algorithms similar to the ones used to compute GVT, or be based on null messages, or even use a combination of the two techniques.

A COMPOSE simulation model can be executed using an optimistic synchronization protocol like Time Warp, a conservative null message protocol, or an adaptive protocol. In the last case, each object specifies whether it will execute in conservative or optimistic mode; further, its execution mode may be changed dynamically. When a model executes in mixed-mode, the EIT of each entity may be computed using existing GVT algorithms or using new asynchronous algorithms based on null messages. Protocols are dynamically switched in COMPOSE by executing the *IAmOptimistic* and *IAmConservative* operations.

Additional operations are provided to control the efficiency of each protocol. For null message protocols, there are operations to specify the lookahead value and the topology. A number of alternatives for checkpointing the state of an optimistic entity are also defined. A C++ object's variables can be contained inside the object or can be on the heap (dynamic variables). An entity can be checkpointed by copying the entire state or by using incremental state saving where the state is compared and only the differences are saved. It is also possible for an object to optionally specify that only a part of its state need be saved when the object is checkpointed.

Detailed performance evaluation of the COMPOSE environment with conservative, optimistic, and adaptive algorithms is in progress.

## 4 COMPLETE EXAMPLE

The following is the complete C++ source for a simple FIFO queue server entity. The server simply accepts jobs (messages) in FCFS order and forwards them to another entity after delaying them by a duration that corresponds to its service time. The *InitMethod* sets up the communication topology for the network for use by conservative and adaptive algorithms.

```
const ServiceTime = 10;
class JobMessageType: public BaseMessageType {};

class ServerEntityType: public BaseEntityType {
 public:
   COMPOSE_EntitySetup(ServerEntityType);
 private:

    class InitMessageType: public BaseMessageType {
     public:
       EntityIDType PredecessorEntity;
       EntityIDType SuccessorEntity;
    };

    // Define entity methods.
    void InitMethod(const InitMessageType& InitMessage);
    void JobMethod(const JobMessageType& JobMessage);

    // Define constructor with BindMethod statements.
    ServerEntityType() {
      BindInitMethod(ServerEntityType, InitMethod,
                     InitMessageType);
      BindMethod(ServerEntityType, JobMethod,
                 JobMessageType);
    }

    // State Variables
    EntityIDType SuccessorEntity;
};//ServerEntityType//

// InitMethod initializes the entity's variables
// and provides the topology and lookahead information
// for conservative or adaptive algorithms.

void ServerEntityType::InitMethod(
      const InitMessageType& InitMessage) {
```

```
  SuccessorEntity = InitMessage.SuccessorEntity;
  AddSuccessorEntity(SuccessorEntity);
  AddPredecessorEntity(InitMessage.PredecessorEntity);
  SetLookahead(ServiceTime);
}//InitMethod//

void ServerEntityType::JobMethod(
    const JobMessageType& JobMessage) {
  Hold(ServiceTime);
  SendMessage(SuccessorEntity, JobMessageType);
}//JobMethod//
```

## ACKNOWLEDGMENTS

The authors gratefully acknowledge use of the Argonne High-Performance Computing Research Facility (128 node IBM SP2). The HPCRF is funded principally by the U.S. Department of Energy, Mathematical, Information and Computational Sciences Division (ER-31).

This work was supported by the U.S. Department of the Air Force/Advanced Research Projects Agency ARPA/CSTO, under Contract F-30602-94-C-0273, "Scalable Systems Software Measurement and Evaluation."

## REFERENCES

Abrams, M. 1988. The object library for parallel simulation (OLPS). In *Proceedings of the 1988 Winter Simulation Conference*, San Diego, CA, 210-219.

Baezner, Dirk, Greg Lomow, and Brian W. Unger. 1990. Sim++: The transition to distributed simulation. In *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, San Diego, CA, 211-218.

Bagrodia, R., W. Liao. 1994. Maisie: A language for the design of efficient discrete-event simulations. *IEEE Transactions on Software Engineering* 20(4): 225-238.

Bryan, Otis. 1989. MODSIM II - an object oriented simulation language for sequential and parallel processors. In *Proceedings of the 1989 Winter Simulation Conference*, Washington, D.C., 172-177.

Gill, D. H., F. X. Maginnis, S. R. Rainier, and T. P. Reagan. 1989. An interface for programming parallel simulations. In *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, Tampa, FL, 151-154.

Jefferson, D. 1985. Virtual Time. *ACM TOPLAS* 7(3):404-425.

Jefferson, D. and H. Sowizral. 1985. Fast concurrent simulation using the time-warp mechanism. In *Distributed Simulation 1985, Society for Computer Simulation Multi-conference*, San Diego, CA, 63-69.

Jha, V. and R. Bagrodia. 1994. A unified framework for conservative and optimistic distributed simulation. In *8th Workshop on Parallel and Distributed Simulation - PADS'94*, Edinburgh, Scotland, 12-19.

Misra, J. 1986. Distributed discrete-event simulation. *ACM Computing Surveys* 18(1):39-65.

Preiss, B. R. 1989. The Yaddes distributed discrete event simulation specification language and execution environments. *Distributed Computing*, 139-144.

Steinman, Jeff. 1991. SPEEDES: Synchronous parallel environment for emulation and discrete event simulation. *Advances in Parallel and Distributed Simulation, SCS Multiconference*, Anaheim, CA, 95-103.

Waldorf, J. and R. Bagrodia. 1994. MOOSE: A concurrent object-oriented language for simulation. *International Journal of Computer Simulation* 4:235-257.

## AUTHOR BIOGRAPHIES

**JAY MARTIN** is a Ph.D. student at UCLA. His research interests include parallel processing and simulation, programming languages and software design and construction.

**RAJIVE L. BAGRODIA** received the B.Tech. degree in Electrical Engineering from the Indian Institute of Technology, Bombay in 1981 and the M.A. and Ph.D. degrees in Computer Science from the University of Texas at Austin in 1983 and 1987, respectively. He is currently an Associate Professor in the Computer Science Department at UCLA. His research interests include parallel languages, parallel simulation, distributed algorithms, and software design methodologies. He was selected as a 1991 Presidential Young Investigator by NSF.