Proceedings of the 1991 Winter Simulation Conference
Barry L. Nelson, W. David Kelton, Gordon M. Clark (eds.)

A TUTORIAL INTRODUCTION TO OBJECT-ORIENTED SIMULATION AND *Sim++*

Greg Lomow
Dirk Baezner

Jade Simulations International Corporation
Suite 80, 1833 Crowchild Trail N.W.
Calgary, Alberta Canada T2M 4S7

Sim++™ is a trademark of Jade Simulations International Corporation

ABSTRACT

The key benefits of object-oriented programming, object-oriented simulation, and object-oriented parallel simulation are presented. This is followed by a tutorial introduction of *Sim++*™, an object-oriented parallel simulation language developed by Jade Simulations.

1 OBJECT-ORIENTED PROGRAMMING

Object-oriented programming is a design and programming discipline that focuses on the objects that make up the system rather than on the overall function of the system. While this is at odds with traditional top-down design techniques, we will see that there are excellent reasons for adopting this point of view.

An *object* is any distinguishable component of a system. An object has a set of *attributes* that defines its state and a set of *operations* for manipulating that state. For example, a lamp is an object. It might have attributes that describe the size of its bulb and whether or not it is on. It might have operations for switching the lamp on and off, and for changing the bulb.

The objects of a system can be divided into groups where each group shares related characteristics. Each group of objects is called an *object class*. Each object is an instance of a particular object class. All instances of the same object class share the same set of attributes and operations.

Object-oriented programming provides a clear set of design guidelines.

1) Identify the object classes that make up the system.

2) Define the interface (i.e., the attributes and operations) provided by each class.

3) Implement the classes.

4) Write the program. It will create and manipulate the objects according to the interface they provide.

Object-oriented programming promotes *mc 'ular design* because each object class encapsulates a data structure and the operations that manipulate that data structure.

Clearly defined interfaces clearly separate the services a class provides from its implementation. This principle is called *separation of concerns* and it has two advantages. First, the implementer can choose any implementation for the class that is consistent with the interface. Second, the user of the class is not affected by modifications to the implementation. Both of these points simplify maintenance.

Clearly defined interfaces also promote *software reuse*. They allow developers to determine whether a class provides the necessary services without having to inspect and verify all of the source code associated with its implementation.

An important technique associated with object-oriented programming is *inheritance*. Inheritance allows a new class to be defined as an extension or refinement of an existing class. The new class is said to be derived from an existing base class. The derived class inherits all of the attributes and operations defined for the base class. The derived class can add new attributes and operations as well as restricting access to the attributes and operations it inherits from its base class. Inheritance promotes modularity because it allows designers to explicitly express the ways in which objects are similar and the ways in which they are different.

Another important technique associated with object-oriented programming is *delayed binding*. Delayed binding is a mechanism that permits the decision about how an operation is to be performed to be deferred from compile-time to run-time. In particular, it allows a derived class to redefine operations provided by its base class and have these redefined operations used everywhere the operation is performed.

Inheritance and delayed binding promote software reuse by allowing a user to take advantage of previously defined classes while still providing

mechanisms for tailoring these classes to specific applications.

2 OBJECT-ORIENTED SIMULATION

Object-oriented simulation and object-oriented programming are both based on the principle that the design of a system should be based on the objects that make up the system. Three concepts characterize the difference between object-oriented programming and object-oriented simulation: entities, events, and simulation time.

In an object-oriented simulation some of the objects are active. That is, they execute independently of, and concurrently with, other active objects. These active objects are called *entities*. Entities are used to model the physical processes in the system being simulated.

An *event* represents a change in the state of one of the objects in the system being simulated. Entities schedule events for each other to mark when these state changes are to occur. Events are used either to synchronize the actions of two entities or to pass information from one entity to another.

The actions of all entities and the scheduling of all events is tied to a logical clock called *simulation time*. Simulation time is an arbitrary, application-defined time scale that is independent of real time. Each event is tied to the logical clock by means of a scheduled event time. This event time corresponds to the actual time in the physical system when the corresponding physical event would occur.

Constructing object-oriented simulations involves:

1) Identifying the physical processes that make up the system being simulated.

2) Defining an entity class to model each type of physical process.

3) Identifying all circumstances that can lead to changes in the state of the system and characterizing these as events.

4) Determining when events occur and tieing them to simulation time by means of their scheduled event time.

3 OBJECT-ORIENTED PARALLEL SIMULATION

An object-oriented parallel simulation is an object-oriented simulation that can be executed in parallel on a multiprocessor. Using multiprocessors allows greater processing and memory resources to be applied to a given problem in a cost-effective manner. These resources can be used to reduce execution time, allow more objects to be simulated, and/or allow more detailed simulations to be performed. Object-oriented simulations are naturally suited to running on multiprocessors because entities represent logically distinct threads of execution that can be executed on separate processors.

Developing object-oriented parallel simulations is facilitated by three factors: determinism, scalability, and portability.

A program is *deterministic* if, given the same input, multiple runs of the program produce identical results. A program is non-deterministic if multiple runs of the same program do not necessarily produce the same results even if they are started with the same input. Almost all distributed parallel programs are non-deterministic. This is because their execution is sensitive to differing processor clock speeds and message latency.

Non-determinism introduced by the run-time system is intolerable in simulations. To understand this, remember that the most difficult phases of any simulation project is validating the model and verifying that the simulation program correctly implements the model. These phases often involve establishing confidence intervals and doing other forms of statistical analysis on the output of the simulation. Non-determinism complicates any statistical analysis by introducing an external source of randomness that cannot be characterized and quantified. This means that the run-time executive for a parallel simulation language is responsible for guaranteeing determinism. To complicate matters, determinism must be maintained no matter how many processors are used to execute the simulation, no matter how the entities are mapped to these processors, and no matter what the architecture of the underlying hardware.

Another critical reason for guaranteeing determinism is that it simplifies debugging and testing. When debugging a non-deterministic distributed program it can be difficult to re-create an error because separate runs can legitimately follow different paths of execution. Determinism ensures that a program follows the same path of execution each time it is run, allowing errors to be easily re-created.

A program is *transparently scalable* if nothing in the program text limits the number of processors that can be used to execute the program. A program is transparently scalable if it can be executed sequentially on a single processor or in parallel on any number of processors without requiring modifications to the program text. Examples of techniques that diminish scalability include:

1) Assumptions about the existence of a central event list.

2) Reliance on shared memory for communication among entities.

3) Restrictions on how entities must be mapped to processors.

Strict object-oriented programming techniques can enhance the extent to which a program is transparently scalable. These techniques forbid objects from communicating using shared memory or from directly accessing and modifying each other's attributes except through well-defined interfaces. Events provide this interface in object-oriented parallel simulation languages.

A program is *portable* if it can be moved among operating systems and hardware platforms without requiring modifications to the program text. The paradigm of entities communicating via events is general enough that it can be mapped to a wide range of machine architectures. Together, transparent scalability and portability permit a parallel simulation to be developed on a workstation in a well supported environment while ensuring that the simulation can be moved, without modifications to the program text, to a multiprocessor for production runs.

4 OBJECT-ORIENTED PARALLEL SIMULATION USING *Sim++*

Sim++ is a C++ package of object types and routines specially designed for writing object-oriented parallel simulations that execute on multiprocessors. C++ is an object-oriented version of C. It is rapidly replacing C as the language of choice for UNIX system applications.

Sim++ programs are deterministic, transparently scalable, and portable across machine architectures and operating systems. Furthermore, *Sim++* provides features that support parallel simulation:

1) *Distributed, parallel input and output facilities.* A major bottleneck in the execution of parallel programs is I/O to and from a multiprocessor. *Sim++* provides facilities for parallel I/O through multiple processors as long as such a mechanism is supported by the underlying operating system and machine architecture.

2) *Built in user level and system level tracing.*

3) *Performance analysis tools.* Achieving high performance results requires tuning the performance of the simulation program. *Sim++* provides tools that estimate the amount of speedup that can be obtained from a particular simulation program. These can be used for evaluating different designs in terms of bottlenecks, granularity, and balance.

4) *Run-time mapping of entities to processors.* Transparent scalability is supported by allowing the mapping of entities to processors to be specified at run-time in a configuration file. This permits the number of processors used for executing a simulation to be easily changed without requiring modifications to the program text.

Sim++ also provides standard simulation libraries for random number generation, data collection, and linked list manipulation.

5 *Sim++* EVENT SETS

Every *Sim++* entity has an event set into which events, scheduled for that entity, are inserted. As an entity selects events for processing, the events are removed from the event set. The process by which events are selected is unique to *Sim++* and requires that an event set distinguish two kinds of events: *future events* and *deferred events.* Future events are those events with an event time greater than or equal to the current simulation time of the executing entity. All events scheduled in a simulation are, initially, future events. Deferred events are those events with an event time less than the current simulation time of the executing entity. A future event becomes deferred if it is left unprocessed in the event set and the entity's simulation time advances beyond the event's scheduled event time. An entity advances beyond the event time of a future event either by advancing its simulation time explicitly, or by selecting another future event with a later event time. Once an event has become deferred, it remains in the event set as a deferred event until selected by the entity.

Sim++ provides a concise set of simulation primitives with which to manipulate an entity's event set. When an entity selects an event using one of these primitives, it specifies a set of application-defined conditions that the event must satisfy. The application-defined conditions are specified using a value known as a *predicate.* An event is selected if it satisfies the given predicate as well as any additional constraints imposed by the primitive.

6 *Sim++* TYPES AND SIMULATION PRIMITIVES

The types and primitives that follow are used in the example presented in the following sections.

sim_entity_id. Associated with each entity in a simulation is an entity identifier that uniquely identifies that entity. Entity identifiers are represented by values of type sim_entity_id. When scheduling an event for an entity, the scheduling entity must specify the entity identifier of the entity for which the event is scheduled. Similarly, when an entity selects an event for processing, it can readily identify the entity that scheduled the event.

sim_predicate. In *Sim++*, an entity is not permitted to directly access its event set. Instead, it must call one of the available primitives and specify an object known as a predicate which is used by *Sim++* to test the attributes of the events in the event set. If an event's attributes satisfy the conditions of the predicate, the event is said to *match* the predicate.

Sim++ provides a number of commonly-used predicates as pre-defined predicate classes, as well as two predicate objects, SIM_ANY and SIM_NONE, that match any and no event, respectively.

sim_time. Values of type sim_time are represented by real numbers greater than or equal to zero. Values of this type are used to represent simulation time.

sim_clock. The primitive sim_clock returns a value of type sim_time that is the calling entity's current simulation time.

sim_schedule. The primitive sim_schedule is used to add future events to an entity's event set. An entity can schedule events for itself or for any other entity that it can identify. The arguments to sim_schedule include the entity identifier of the entity for which the event is scheduled, a simulation time delay (the scheduled event time of an event is the sum of the current simulation time and the delay), an event type, and an event body and length. The body of an event may contain any arbitrary data (e.g., a text string, a C++ object, or even an array).

sim_hold. The primitive sim_hold is used to model a simulated delay that can be interrupted by events. The arguments to sim_hold include a simulation time delay that specifies the duration of the hold, a predicate that specifies the application-defined conditions an event must satisfy in order to interrupt the delay, and a reference to an event structure in the calling entity's state in which *Sim++* will return the event, if any, that interrupted the simulated delay. sim_hold can be interrupted by any future event that satisfies the given predicate and whose event time is less than the end time of the simulated delay. If interrupted, sim_hold returns the number of simulation time units remaining of the original delay; otherwise, sim_hold returns 0.0 (i.e., the entire delay expired without interruption).

sim_wait. The primitive sim_wait is used to select a future event satisfying a given predicate. The arguments to sim_wait include a predicate that specifies the application-defined conditions an event must satisfy to be selected from the calling entity's event set, and a reference to an event structure in the calling entity's state in which *Sim++* will return the selected event.

sim_select. The primitive sim_select is used to select a deferred event satisfying a given predicate. The arguments to sim_select include a predicate that specifies the application-defined conditions an event must satisfy to be selected from the calling entity's event set, and a reference to an event structure in the calling entity's state in which *Sim++* will return the selected event.

sim_waiting. The primitive sim_waiting is used to count deferred events satisfying a given predicate. The argument to sim_waiting is a predicate that specifies the application-defined conditions an event must satisfy in order to be included in the event count returned by sim_waiting.

7 TUTORIAL EXAMPLE: A TRANSPORTATION SYSTEM

The rest of this paper presents a tutorial intended to give the reader a brief overview of *Sim++*. A simple transportation system simulation is used to introduce the basic simulation concepts and facilities provided by *Sim++*.

Consider a transportation system linking two cities situated on opposite sides of a river. A vital part of this transportation system is a ferry system in which a single ferry travels between terminals located on opposite sides of the river. Vehicles travel from a city to the ferry terminal located on their side of the river where they wait for the ferry to transport them to the terminal on the opposite shore. Once across, the vehicles travel from the ferry terminal to the city on that side of the river. This structure is illustrated in Figure 1.

For this example, the cities and the ferry system are modelled as entities and the vehicles as events passed among these entities. Alternatively, the vehicles could be modelled as entities as well, but this approach has not been been adopted here.

7.1 Deriving New Entities

Sim++ provides a special class of C++ object called sim_entity which serves as the base class from which new entity types are derived. For each type of

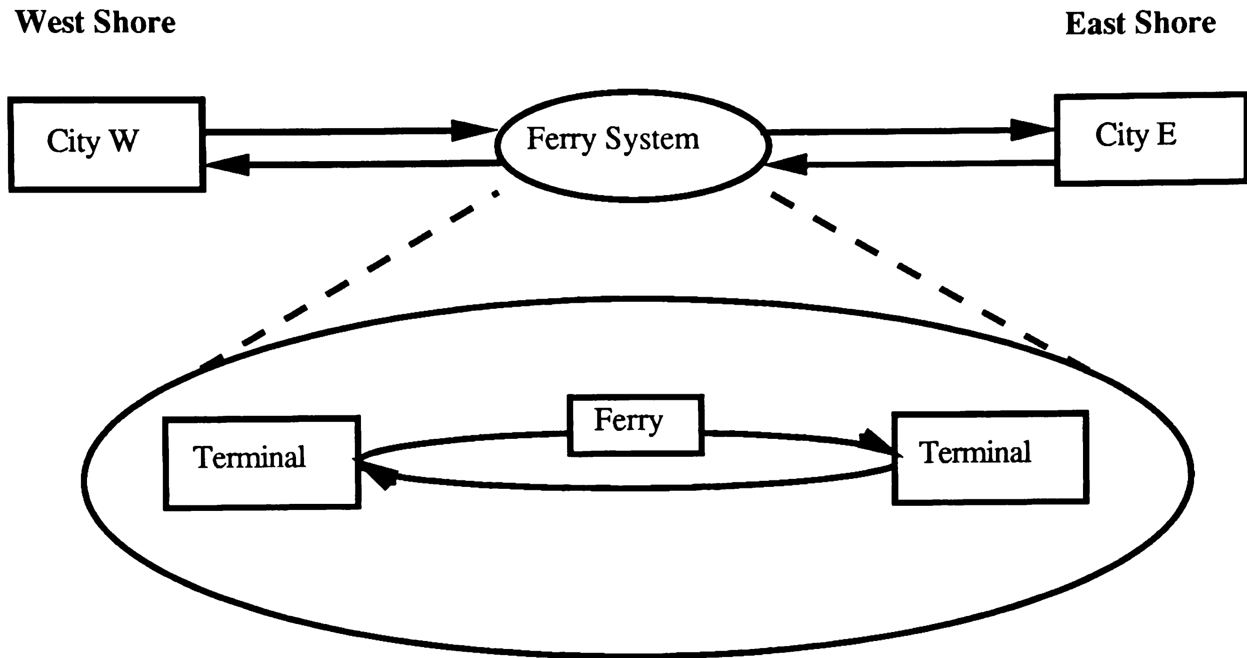

Figure 1: A Transportation System

active component in a simulation, the user derives an extension of class sim_entity. The ferry example requires two such **entity class declarations**. In the examples that follow, an ellipse (i.e., ...) is used, ungrammatically, to indicate that C++ code has been omitted for brevity. The symbol // is used in C++ to denote the beginning of a comment that continues until the end of the current line.

```
class city : public sim_entity {
public:
  void body(); // main body of class city
};

class ferry : public sim_entity {
public:
  void body(); // main body of class ferry
};
```

In the example shown above, class city and class ferry are both derived as extensions of class sim_entity. Each entity class declaration must include a function declaration for its **main body**.

The main body for a class derived from class sim_entity serves to define the actions of that type of active component. When an entity executes, its actions are defined by its main body. Skeletons of these actions for city and ferry are shown next.

```
void city::body()
{ ... // initialization
  while (sim_clock() < Duration) {
    ... // schedule an event for the ferry
    ... // hold for inter-arrival time
    ... // receive unloaded vehicles
  }
  ... // final actions
}

void ferry::body()
{ ... // initialization
  while (sim_clock() < Duration) {
    ... // load vehicles waiting for transport
    ... // hold for crossing time
    ... // unload vehicles
  }
  ... // final actions
}
```

In the code fragments shown above, the initialization of the two entity types might involve, for example, the initialization of data structures or variables required by each entity, as well as the exchange of events between the cities and the ferry in order to synchronize these entities at the beginning of the simulation. Similarly, the final actions of each entity might again involve the exchange of events to collect and summarize statistics.

7.2 Class City

The entity class declaration for class city is shown next.

```
class city : public sim_entity {
  sim_time Duration;
  sim_entity_id Ferry;
  sim_normal_obj Driving;
  sim_negexp_obj Next_vehicle;
  sim_time Delay;
  sim_event Ev;
  int Count;
public:
  void body();
};
```

In addition to the declaration of the main body for class city, this entity class declaration includes several member variables which constitute the state of this entity.

- Duration - length of the simulation.
- Ferry - identifier of the entity modelling the ferry system.
- Driving - a random number object used to generate the times necessary to drive from the city to the ferry.
- Next_vehicle - a random number object used to generate the intervals between successive vehicles leaving this city for the ferry.
- Delay - time remaining until the next car leaves this city for the ferry.
- Ev - used to hold events representing cars arriving from the ferry.
- Count - maintains the number of cars that have arrived at this city from the ferry.

The main body for class city is shown below.

```
void city::body()
{ ... // initialization
    while (sim_clock() < Duration) {
      // hold until time to generate next car
      Delay = sim_hold(Next_vehicle.sample(),
                       SIM_ANY, Ev);
      while (Delay > 0.0) {
        // count vehicles arriving with ferry
        Count = Count + 1;
        // hold until time to generate next car
        Delay = sim_hold(Delay,SIM_ANY,Ev);
      }
      // schedule arrival of vehicle at ferry
      sim_schedule(Ferry, Driving.sample());
    }
    ... // final actions
}
```

The outermost while loop ensures that the city continues to execute as long as its simulation time is less than the duration of the simulation.

Each time through this loop, the city generates one new vehicle. It does this in three steps: 1) it generates the delay until the next vehicle will leave the city and waits for this amount of time to elapse; 2) if this interval is interrupted (i.e., Delay > 0.0), then the city records the fact that a vehicle has arrived at this city after being unloaded from the ferry and holds for whatever time is remaining until the next vehicle is to be generated; and 3) eventually the hold will complete without being interrupted and the city will schedule an event for the ferry that represents the arrival of a vehicle at the ferry.

The main body for city makes use of two random distribution objects. The first is Driving which is used to generate the amount of time required to drive from the city to the ferry. The second is Next_vehicle which is used to generate the interarrival delay between successive vehicles.

The calls to sim_hold use a special, pre-defined predicate value called SIM_ANY. This predicate will match any event against which it is compared (i.e., the hold may be interrupted by *any* future event with an event time less than the end of the simulated delay).

7.3 Class Ferry

The entity class declaration for class ferry is shown below.

```
class ferry : public sim_entity {
  sim_time Duration;
  int On_board;
  sim_event Ev;
  sim_entity_id East, West, Side;
  sim_normal_obj Loading, Crossing,
                 Unloading, Driving;
public:
   void body();
};
```

In addition to the declaration of the main body for class ferry, this entity class declaration includes several member variables which constitute the state of this entity.

- Duration - length of simulation
- East, West - identifiers of the entities modelling the cities on the east and west shores of the river, respectively.
- Side - identifier of the city on the side of the river on which the ferry is currently waiting; initially the ferry is on the east side of the river.
- On_board - a count of the number of vehicles currently on the ferry; initially the ferry is empty and On-board is 0.

- Loading, Crossing, Unloading, Driving - random distribution objects for generating independent streams of random numbers.
- Ev - used to hold events returned by Sim++.

The main body for class ferry is shown below.

```
ferry::body()
{ ... // initialization
  On_board = 0;
  Side = East;

  while (sim_clock() < Duration) {
    // load up to 4 vehicles
    while (On_board < 4 &&
           sim_waiting(sim_from_p(Side)))
    {
      sim_hold(Loading.sample(),
               SIM_NONE, Ev);
      On_board = On_board + 1;
      sim_select(sim_from_p(Side), Ev);
    }

    // cross to opposite shore
    sim_hold(Crossing.sample(),
             SIM_NONE, Ev);
    if (Side == East)          // switch sides
      Side = West;
    else
      Side = East;

    // unload vehicles
    while (On_board > 0) {
      On_board = On_board - 1;
      sim_hold(Unloading.sample(),
               SIM_NONE, Ev);
      sim_schedule(Side, Driving.sample());
    }
  }
  ... // final actions
}
```

The outermost while loop ensures that the ferry continues to operate as long as its simulation time is less than the duration of the simulation. Inside of this loop the ferry engages in three distinct sets of actions.

First, the ferry loads vehicles as long as it is not full and there are vehicles waiting on this side of the river. Loading a vehicle involves waiting for a random amount of time to elapse, incrementing the count of vehicles on board the ferry, and removing the event representing that vehicle from the event set.

Second, the ferry crosses the river. This involves waiting for a random amount of time to elapse and then updating the value of Side. The ferry uses Side to indicate which shore it is on. Side alternates between the values of East and West, two entity identifier variables that uniquely identify the two city entities.

Third, the ferry unloads each vehicle it has on board. This involves decrementing the number of vehicles on the ferry, holding for a random amount of time to elapse, and then scheduling an event for the city on this side of the river announcing the arrival of this new vehicle.

All calls to sim_hold in the ferry use a special, pre-defined predicate value called SIM_NONE. This predicate will never match any event against which it is compared (i.e., the hold cannot be interrupted by any events). Therefore, all events scheduled for the ferry by either city always become deferred before being selected. In calls to sim_waiting and sim_select, the ferry uses a special type of predicate called sim_from_p, passing it an entity identifier as its only parameter. Predicates of this type will only match deferred events that were scheduled by the entity identified in the parameter. This means that if the ferry has reached the eastern shore, for example, it will only load vehicles waiting on that shore; events scheduled by the city of the opposite shore remain deferred.

AUTHORS' BIOGRAPHIES

GREG LOMOW is Director of Product Management for Jade Simulations International Corporation. He received his M.Sc. and Ph.D. in Computer Science from the University of Calgary. His current research interests are simulation languages, distributed systems, parallel simulation languages, and programming environments.

DIRK BAEZNER received his M.Sc. in Computer Science from the University of Calgary in 1991 for his thesis, *Language Design for Parallel Simulation*. For the last three years, he has worked in Jade Simulation's product development division, where he co-designed and implemented *Sim++* for Jade's TimeWarp system. His current research focuses on the design of parallel simulations and parallel simulation languages.

Both authors can be reached at:

Jade Simulations International Corporation
Olympic Volunteer Centre, McMahon Stadium
#80, 1833 Crowchild Trail, N.W.,
Calgary, Alberta, Canada T2M 4S7
(403) 282-5711