SUPPORTING DISTRIBUTED OBJECT-ORIENTED SIMULATIONS IN AN OPEN PLATFORM

Carlos A. Maziero
Computer Science Dept.
Catholic University of Paraná
80.215-901 – Curitiba – Brazil
maziero@ppgia.pucpr.br

Richard D. Ribeiro
Computer Science Dept.
CEFET-PR
80.230-901 – Curitiba – Brazil
richard@dainf.cefetpr.br

Keywords

Discrete simulation, object-oriented models, distributed simulation.

Abstract

This work aims to define the structure of a distributed discrete-event simulation runtime support. To simplify the construction of simulation models, we chose to use object-oriented languages, thus allowing hierarchical construction of the models, and code reuse. Instead of creating a model description language specific to this simulation support, we used well-known object-oriented simulation libraries, giving us a more generic and portable environment. The problems arising from the use of object-oriented programming models over a heterogeneous distributed computing environment are solved by the use of an open platform for object communication, according to the CORBA standards.

1 Introduction

The use of computer simulation techniques can be particularly interesting in the case of complex systems or systems having a large number of entities. Within that sort of systems the use of an analytical approach has shown to be difficult (Chandy and Misra, 1981; Righter and Walrand, 1989). However, the simulation of complex systems can represent a difficult task, in terms of computational effort (Misra, 1986).

In general, the use of parallel processing can significantly reduce the execution time of large simulations, because the models involved normally present a fair amount of potential parallelism, *i.e.* actions that can be done in an almost independent manner. The main problem in building a parallel simulation is to provide efficient solutions that respect the causality relations present in the simulation model, thus ensuring the validity of the simulation results. Some synchronization mechanisms should be implemented to ensure that causality will not be violated (Misra, 1986; Righter and Walrand, 1989; Fujimoto, 1990).

The construction of simulation models of complex systems represents another problem to be solved. The object-oriented programming approach can be very useful in the construction of such models, as it allows a more intuitive translation from the system entities and their relationships into a computational model. The object orientation paradigm has its own origins in the simulation area, by means of the simulation language *Simula*, which first introduced the concepts of class, object, attribute, and method (Birtwistle et al., 1973).

This work aims to define the structure of a distributed discrete-event simulation runtime support. To simplify the construction of simulation models, we chose to use object-oriented languages, thus allowing hierarchical construction of the models, and code reuse. Instead of creating a model description language specific to this simulation support, we used well-known object-oriented simulation libraries, giving us a more generic and portable environment. The problems arising from the use of object-oriented programming model over a heterogeneous distributed computing environment are solved using an open platform for object communication, according to the CORBA standards (OMG, 1995).

This article is organized as follows: section 2 presents the main concepts and techniques used in sequential discrete-event simulation, the problems involved in running such simulations in a distributed execution environment, and their solutions; section 3 introduces the use of object orientation to build simulation models and shortly presents the CORBA architecture; finally, section 4 presents the definition of an open platform to run distributed discrete-event simulations of object-oriented models.

2 Discrete-event simulation

In a simulation, the time values considered in the model can evolve at a different rate from the real (physical) time. This abstract notion of time is usually called *simulation time*, or *virtual time*. Its independence from the real time allows to execute the simulation at rates compatible with the analysis to be done on the model being simulated.

The mechanisms that coordinate the simulation time evolution should ensure two basic principles: *causality*, by which the future of a system cannot change its past, and *determinism*, by which the future states of a system can be determined from its present an past states.

In discrete-event simulations, the order in which events are processed should respect existing causality constraints, to ensure that each event will happen only after the events it depends on. To allow this, sequential simulation mechanisms use a scheduler, that can be viewed as a queue in which events are ordered by increasing execution time. The first event in the scheduler is the next one to be processed, and its execution time corresponds to the present in the simulation time. The processing of this event can generate new events in the future, that are inserted in the scheduler queue (maintaining the time order). It can also cancel events already present in the scheduler queue. This means that the events in the scheduler queue are potential events, as they can be modified or canceled by the execution of the first event.

2.1 Distributed simulation

The distributed execution of a discrete-event simulation over a set of processors can increase the overall speed of the simulation, but the simulation mechanisms should be capable of efficiently explore the potential parallelism present in the simulation model.

The main problem in the distributed execution of a simulation is the compromise between the maximal use of the parallelism present in the model and the respect to the causal constraints. Several works have been done to propose solutions to this problem (Chandy and Misra, 1979; Chandy and Misra, 1981; Misra, 1986; Righter and Walrand, 1989; Fujimoto, 1990).

To present the synchronization methods used in distributed simulations, we can consider a simulation model based on the process/message paradigm. The models are then composed of a static set of processes that communicate by messages sent over reliable FIFO channels (figure 1). With this model structure, each process can be considered as an almost independent sequential simulator, responsible for the simulation of an entity in the model, and using messages to interact with the other processes, sending and receiving events.

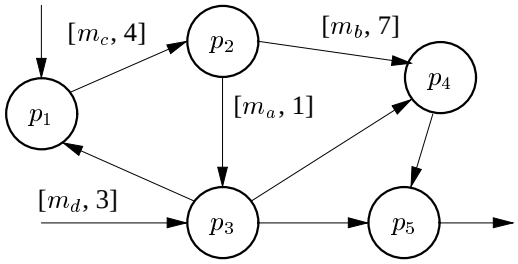

Figure 1: Simulation Model

To assure the simulation correctness, all processes in the model should consider the same logical time reference (i.e. a unique global simulation clock). However, the use of a global clock leads to a completely synchronous simulation, ignoring the potential parallelism present in the model. To efficiently use this parallelism, an asynchronous approach should be employed to drive the simulation execution.

To allow an asynchronous evolution of the simulation, each process p_i should manage a local copy of the global simulation clock, which is called *local clock* or *local time* (tl_i). As processes can have different local clocks, some synchronization mechanisms should be defined to assure the simulation correctness, i.e. the causality principle (Righter and Walrand, 1989).

Chandy and Misra (Chandy and Misra, 1981) showed that, if each process locally respects the causality principle and the channels are FIFO, then causality will be respected by the whole simulation. To preserve local causality, each process should treat its local events (internal events and events received from another processes) respecting their increasing execution dates. To allow this control, every process should timestamp all messages it sends, using its local simulation time.

Using this schema, each process should consider the messages it receives in their timestamp order. But normally a process has no means to know the timestamps of the messages it will receive in the future, and then constructing such an order for the future local events can be impossible.

Two classes of solutions were proposed to solve this problem (Fujimoto, 1990):

- *Pessimistic (or conservative) approach*: We should assure that all local events will be treated in the correct order, thus respecting the causality principle all the time. If a process cannot decide about the next event to execute, it

is suspended until this decision becomes possible. As processes can be suspended to wait for events from other processes, additional mechanisms should be used to avoid the occurrence of deadlocks, or to solve them.

- *Optimistic approach*: The process take the (optimistic) assumption that it will receive no more events from other processes, and then it is able to decide about the next local event to execute. If the process receives a message later, with a timestamp smaller than its local clock, a local causality violation is detected. The local simulation should then be backtracked to the date of the received event and be re-executed considering the new event. Some mechanisms are needed to save and restore past states and to cancel messages sent incorrectly.

The pessimistic approach needs mechanisms to deal with deadlocks. A classical solution to avoid deadlocks was proposed in (Chandy and Misra, 1979). It consists in using *null* messages to avoid deadlock situations. The *null* messages have no meaning to the simulation model itself and carry only their own timestamps. A process uses null messages to inform to other processes "forecasts" about the next events it can send to them. Thus, a process p_i at a logical time tl_i uses a null message $\{null, tl_i + \delta\}$ to inform other processes that the next events it can send will be dated at least $tl_i + \delta$. This schema works based on the assumption that channels are FIFO and processes act as sequential simulators. The value of the forecast δ, also called *lookahead*, is normally defined by the process using its local data, as the logical time needed to treat an incoming event, the next event date in its scheduler, etc.

Each time a process receives a null message, it can verify if some local event can be treated, i.e. if the next local event can be defined. Also, the process can re-evaluate its own lookahead and inform other processes if it has changed.

3 Object-Oriented Simulations

The use of object-oriented programming in the construction of discrete-event simulations can be considered simple and intuitive. Using this approach, real entities are modeled as objects created from classes that define their characteristics and behavior. Interactions between these entities can be modeled as methods invocations between the corresponding objects.

As in real world situations there are active entities (like robots, processors, persons, etc) and passive ones (like buffers, network packets, etc), a simulation model can include active and passive objects. The possibility to have more than one active object in a simulation leads to the need of concurrency support in the simulation environment, using mechanisms like threads, semaphores and so on. The active objects in a sequential object-oriented simulation are under the control of a scheduler, to ensure all the causality constraints.

3.1 Distributed object-oriented simulation

Besides the basic mechanisms needed to manage discrete-event simulations, the construction of an object-oriented simulation environment over a heterogeneous distributed context implies:

- to implicitly map simulation models expressed as objects and methods to execution models more appropriate to be executed in a distributed environment, such as a model based on processes and messages;

- to distribute objects on the machines in which the simulation will be run, to balance the workload among them and to minimize the communication needs;

- to provide communication support between objects located in different memory spaces, preserving the method invocation semantics, and, if possible, hiding the physical separation between the objects and the differences between the hardware, operating systems, and languages involved in the communication.

- to synchronize the active objects execution, to make logical time advance, without violating the causality constraints.

There are proposals of object-oriented environments for distributed simulation, some of them based on proprietary languages, like *MOOSE* (Waldorf and Bagrodia, 1994), *SIM++* (Lomow and Baezner, 1991), and others based on libraries on a known language, like *COMPOSE* (Martin and Bagrodia, 1995) and *PROSIT* (Mallet and Mussi, 1994). These environments do not consider the heterogeneity aspects of the computing platform, or take them into account using proprietary solutions.

The work presented in this paper aims to define an object-oriented discrete-event simulation environment on top of an open distributed platform. To achieve this, we adopted a distributed objects CORBA infrastructure as the basis layer of our simulation environment.

3.2 The CORBA Architecture

The *Common Object Request Broker Architecture* is an infrastructure for distributed object programming proposed by the *Object Management Group* Consortium (OMG, 1995). It provides interoperability between object-oriented applications independent of machine hardware, communication protocols, operating systems and even object languages. The architecture is built on top of a nucleus called ORB – *Object Request Broker*, which is responsible for all the interactions between objects. The other services provided by the CORBA framework are implemented as CORBA objects, which interact and can be accessed using the ORB. The figure 2 presents a simplified view of this architecture.

To be able to communicate, objects should have their interfaces (public methods and attributes) defined and published using an interface definition language (the CORBA IDL). The compilation of an IDL interface definition file generates the communication stubs needed to access objects that implement such an interface.

The IDL interface definitions can also be converted to a binary inter-operable format, and stored in an *interface repository*, where they can be accessed at any time. Using the DII - *Dynamic Invocation Interface*, objects that have no access to the static communication stubs of a given interface can get access to its binary description, discover its structure, and dynamically build method invocations on objects which implement that interface.

On the reception of a method invocation request for a given object, the ORB locates the corresponding implementation code and transfers the call parameters to it, using the BOA – *Basic Object Adapter*. When the requested method execution is finished, the return values and any exceptions are returned to the caller object. For both objects the method call is accomplished transparently; the ORB is in charge of locating the target object, coding the parameters in an architecture-neutral format, marshaling them, and doing the inverse work on the target side.

4 The proposed simulation environment

In this section we describe the architecture of the system we proposed for supporting distributed object-oriented discrete-event simulations. For the execution support, we are considering a set of computing nodes connected by a local network. The network provides the only communication support between the computers, as there is no shared memory space. All the components of this system are assumed to be reliable, thus we do not consider fault tolerance aspects in this work. On top of this basic structure, a CORBA platform gives the communication facilities needed to distributed object interactions.

4.1 System Architecture

We have a simulation model, composed by a set of simulation objects, to be executed on a group of computing nodes. Each node should then receive and execute a subset of the simulation model objects. To optimize the simulation time management, all the objects situated at the same node will be under the control of the scheduler. Thus, each computing node will contain a sub-simulator, responsible for the execution of the local set of objects. Sub-simulators will interact to exchange synchronization information and to transfer method invocations between the model objects. Interactions between sub-simulators will be done using the services offered by the CORBA platform. The figure 3 presents this structure.

Each sub-simulator executes a sequential simulation, considering its own objects and the events (method calls) coming from other sub-simulators. These external events should be considered correctly to avoid causality problems. As we will show later in this text, respecting some conditions each sub-simulator can implement its local sequential simulation using any simulation library or language.

Normally each sub-simulator is implemented as a unique process. This approach aims to minimize context switches and to allow a better integration with the sequential simulation libraries used. As there is only one simulation clock for all the objects in a sub-simulator, these objects can interact not only using method calls but also using shared objects, with no risk to the causality constraints. Interactions between remote objects remain, however, restricted to method invocations, due to the asynchronous execution of the sub-simulators.

Each sub-simulator has under its control the local model objects and some additional objects for the management of both the distributed synchronization and events exchanges. These management objects should be considered (scheduled) in the same way the simulation objects are, to simplify the sub-simulator internal structure and to maintain its portability. However, this assumption should not sacrifice the simulation performance.

An object-oriented sequential simulation library serves as the basis for the construction of each sub-simulator, offering the basic mechanisms needed for the management of the local simulation. The man-

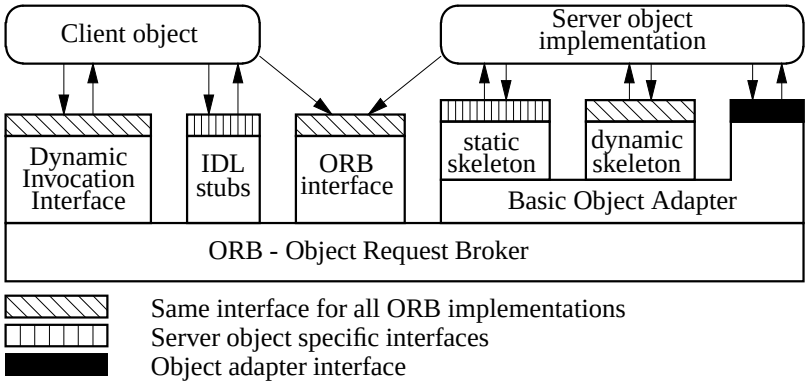

Figure 2: The CORBA architecture

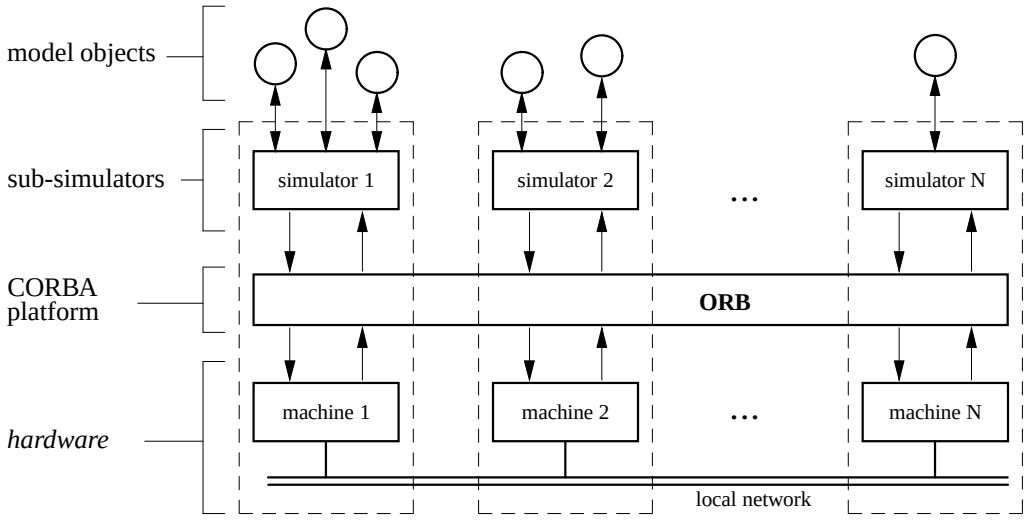

Figure 3: System architecture

agement objects are dealt with the model objects; therefore they should use only common services offered by the simulation sacrifice, as inserting or removing events from the scheduler, reading the local clock, etc. This allows the library to be easily substituted if needed. In our prototype we are currently using the C++ simulation library *C++SIM* (Little, 1994).

The sub-simulators interact using the CORBA platform services. We use relatively few services of this platform, mainly to provide transparent communication between remote management objects. Interactions between local objects inside a sub-simulator can be done without using the ORB, helping to improve the system performance. Therefore, a sub-simulator should only define its IDL interface concerning the management and event exchange services.

In the next sections we present the internal structure of the sub-simulators and the interactions between them.

4.2 Sub-simulator architecture

Each sub-simulator is composed of:

- *Sequential simulation mechanisms*, basically the scheduler and its associated services. These functionalities are provided by the sequential simulation library employed.
- *Local simulation objects*, that represent the subset of the model allocated to that machine.
- *Synchronizer object*, which should collect information about the local simulation (local clock, scheduled events, lookaheads, etc) and cooperate with the synchronizer objects situated on the other sub-simulators in order to implement a global synchronization strategy. This

object should also control the evolution of the local clock, according to the global synchronization strategy.

- *Event dispatcher object*, which main work is to send and receive events (method invocations) to and from remote sub-simulators. It timestamps all the events sent, notifies the local synchronizer object about sent and received events and locally schedule the received events to be activated in their timestamp dates.

Figure 4 shows the internal structure of a sub-simulator, including the elements described here and their main interactions.

The sub-simulation kernel comprehends the scheduler mechanisms provided by the sequential simulation library. All the local objects, including the synchronizer and the event dispatcher objects, have their execution managed by the scheduler. The sub-simulator IDL interface, that should be defined to allow it to be accessible using CORBA, is defined by the external services provided by the synchronizer and event dispatcher objects. We will see in the following sections these objects and their interfaces in greater detail.

4.3 Event dispatcher

Using this framework, we can have two distinct situations for method invocations. If the communicating objects are on the same sub-simulator, they are under the control of the same scheduler, hence they have the same simulation time reference. In this case the method call can be done directly, with no external management.

In the case where the objects are situated in different sub-simulators, the method invocation request should be time-stamped and sent to the sub-simulator where the remote object is located. Upon the receipt of the method call, the remote sub-simulator schedules it locally for execution in the timestamp date, to respect causality constraints.

The activities related to the method invocation requests transfers between sub-simulators are managed by the *event dispatcher* object. Its main functions are:

- send and receive, through the ORB, method invocation requests. These requests can be considered as tuples {*time, source, target, method, parameters*};
- timestamp all sent requests with the current local simulation time;
- schedule the received requests in the local simulation scheduler, considering the request timestamp;
- notify the synchronizer object about all requests sent and received, with their respective dates, origins, and destinations.

Some remarks about the implementation of these operations should be given. The event dispatcher object is in charge of scheduling the received events in the local scheduler. However, normally the simulation libraries only allows an object to schedule its own execution, and not other objects' executions. This leads us to the following implementation for the the event reception service:

```
method dispatcher.event_receive
       (date,source,target,method,params)
begin
   // notify the local synchronizer object
   synchronizer.notify_reception (date,source);
   // schedule itself to tl = date
   hold (tl = date);
   // execute the method invocation request
   call target.method (params);
end
```

In the code above we observe that the dispatcher object suspends itself in the scheduler until the event received can be executed (`tl` $\geq$ `date`). During this period in which the dispatcher is scheduled to execute a received event, new method invocation requests can arrive from outside, containing possibly different (and even smaller) execution dates. These new events should also be scheduled as soon as they're received, to be considered in the local simulation.

This leads to the necessity of creating an independent thread for each received event, to allow the dispatcher to receive and schedule several events simultaneously. Thus, the operating system, the CORBA platform and the simulation library used should all support multi-thread programming. If this condition is not respected, all incoming events will be treated one at a time, serially, leading to poor performance and deadlock risks[1].

To simplify the implementation, we are only considering asynchronous object requests between remote objects, using the CORBA `one-way` method invocation semantics. Local method invocations can use synchronous or asynchronous method invocations.

4.4 Sub-simulators synchronization

To allow the evolution of the whole simulation, sub-simulators should interact with each other to maintain consistency between their local clocks and to ensure causality constraints between events. This can

[1] In a conservative synchronizer, the incoming events timestamps are used to update the local time and the lookaheads, to allow the execution of the event currently scheduled by the dispatcher. If incoming events are not taken into account, the local sub-simulator can remain blocked forever.

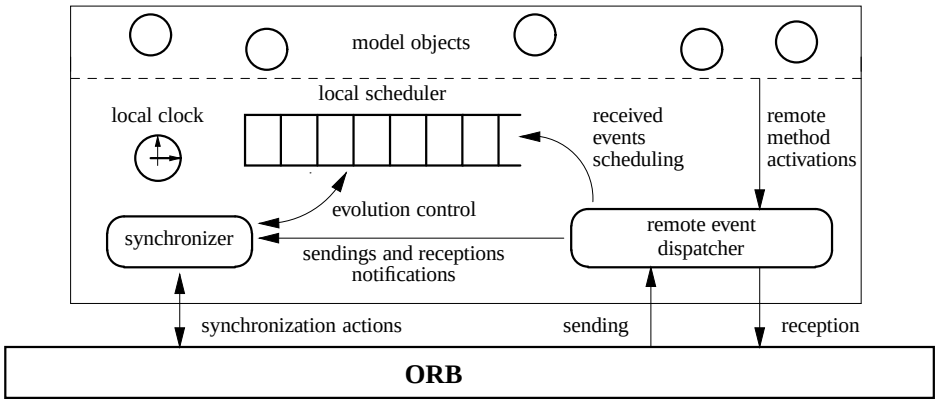

Figure 4: Sub-simulator internal structure

be achieved by using a synchronization strategy like those presented on section 2.1.

The objects responsible to manage global synchronization tasks are the *synchronizer objects* situated in each sub-simulator. These objects should collect local information and interact with each other to implement a pessimistic or optimistic synchronization strategy, acting on the local schedulers as needed to ensure the correct evolution of each local simulation.

Although the functions realized are equivalent, pessimistic and optimistic synchronizer object implementations are based on very distinct concepts. We will present each approach separately in the following.

4.4.1 A pessimistic synchronizer

We will present the structure and behavior of a pessimistic synchronizer implementing the null-messages deadlock avoidance technique (Chandy and Misra, 1979; Fujimoto, 1990). Such a pessimistic synchronizer should:

- Manage channel clocks: each input and output channel of the sub-simulator has a clock associated to it. This clock should contain the timestamp of the last message sent or received on that channel. The minimum of all input channels' clocks is called the *input clock* (ti), and indicates the minimum timestamp of the next event to be received from other sub-simulators.
- collect information on the local model objects to update local lookaheads;
- send and receive null-messages, to propagate local lookaheads and to update channel clocks;
- coordinate the local simulation evolution, controlling the scheduler execution in order to avoid causality violations. Mainly, the local simulation clock should never be greater than the local input clock, due to the risk of "jumping over" possible external events without considering them.

The management of the channel clocks (and consequently of the input clock ti) can be implemented using the information about the sending and reception of event requests given by the dispatcher object (section 4.3). As sending and reception notifications have respectively the forms `[date, target]` and `[date, source]`, channel clocks management can be done easily.

The lookahead values needed to implement this synchronization technique can be given by the model objects to the synchronizer using an appropriate method call; they should use this method to periodically update their lookaheads. Automatic methods for lookahead calculation are not in the context of this work, being presented in (Fujimoto, 1988; Mehl, 1991). Null -messages sending is done by comparing the output channel clocks and the local lookaheads of the next event to be sent on each output channel. All the null-messages received are used just to update their respective input channel clocks, being dropped after it.

To coordinate the local scheduler evolution with the input clock evolution, we make use of a simple but efficient approach. As the synchronizer is an active object which execution is controlled by the scheduler, the synchronizer object asks continuously to be scheduled at the date corresponding to the current input clock value. Therefore, this object will "monopolize" the scheduler, only releasing it to other events when their scheduled dates are older or equal than the current value of the input clock. This schema will only allow the execution of events that presents no risk for the global causality constraints.

Based on these descriptions, we can show the basic structure of a pessimistic synchronizer object and its methods:

```
object synchronizer
begin   // main loop
   repeat
      // synchronize with others
      send_null_messages ();
      // update input clock ti
      ti:= min_i(input[i].clock);
      // reschedules for next execution
      hold (tl = ti);
   until end_of_simulation ;
end

// notify event reception to synchronizer
method notify_reception (date,source)
begin
   if input[source].clock > date then
      error: causality violation !
   else
      input[source].clock := date ;
   endif
end

// notify event sending to synchronizer
method notify_sending (date,target)
begin
   if output[target].clock > date then
      error: causality violation !
   else
      output[target].clock := date;
   endif
end

// send null messages
method send_null_messages ()
begin
   lookahead := tl + lookahead;
   for each target T do
      if output[T].clock < lookahead then
         output[T].clock := lookahead ;
         send {null,lookahead} to T;
      endif
   end
end

// update local lookaheads
method update_lookaheads ()
begin
   ...
end
```

This strategy makes the coupling between the synchronizer object and the local scheduler completely transparent, hence the scheduler code does not need to be modified. Furthermore, this strategy adjusts itself automatically to the workload present in each sub-simulator. Thus, if the local scheduler has many local events to process, the synchronizer scheduling will be done less frequently, to make local simulation progress faster. On the other hand, if few local events are scheduled, the synchronizer object will be activated more often, so any updates in the input clock (by the arrival of null messages) will be taken into account quickly.

4.4.2 An optimistic synchronizer

The simulation architecture presented can accept different synchronization strategies, but some technical difficulties can be found when implementing optimistic strategies, mainly caused by the backtracking mechanisms. In an optimistic approach (section 2.1), when a causality error is detected, the synchronizer should be able to interact with the local scheduler and model objects to restore their state to a previous date, and to cancel events incorrectly dispatched to other sub-simulators. Let's examine how each of these tasks can be accomplished:

- *Save and restore object states*: this task can be easily done by using state transfer services that should be implemented by each model object. A similar approach was used in the Isis system (Birman and A., 1985) to update states between process replicas. Following this, each object should implement two methods: *GetState* and *SetState*. Both methods can be called by the synchronizer, the first one to get the current state of the process and the second one to set its state to a previous saved state. The synchronizer should then manage a list of past states for each local model object.

- *Cancel events sent incorrectly*: the synchronizer is notified by the dispatcher object of each event sent to another sub-simulator. Thus it is able to manage a list of events sent, to be used for possible event cancellations. All the local interactions are canceled restoring the local objects to their past states.

- *GVT calculation*: the *Global Virtual Time* value represents the minimum time value of all pending events in the whole simulation. This value is used for the management of backtracking information. States and messages older than the GVT value can be safely discarded to free memory, because they are no more useful. This value can be seen as a global stable property, and can be easily calculated with classical distributed algorithms (Raynal, 1992).

- *Backtrack the local scheduler*: this task can present some difficulty, mainly because some modifications on the scheduler code certainly will be needed. Depending on the complexity and code availability of the simulation library, the construction of such an optimistic synchronizer can become unfeasible.

Thus we can conclude that the implementation of an optimistic synchronizer can be significantly more complex than its pessimistic counterpart, and then can limit the choice of sequential simulation library used to provide the basic simulation mechanisms needed locally.

5 Conclusion

This work presented the architecture of a distributed environment to support object-oriented discrete-event simulations over heterogeneous platforms. This structure aims to cover some aspects of simulation environments, like the support of object-oriented models, and the execution over a distributed heterogeneous middleware.

As shown, our proposal allows the integration of different sequential simulation libraries, notably if a pessimistic synchronizer is used. In fact, the interaction schema between the synchronizer and the local scheduler is simple but powerful, as the synchronizer presents itself just as another object to be scheduled. Allied to the ORB, this feature allows to integrate different object-oriented simulation libraries, languages and platforms.

We presented the implementation of a pessimistic synchronizer based on the null-messages deadlock avoidance technique. Nevertheless the structure has enough flexibility to easily allow other kinds of pessimistic synchronizers. It is also possible to define synchronizer classes specialized in each strategy, as it is done in *MOOSE* (Waldorf and Bagrodia, 1994) and *PROSIT* (Mallet and Mussi, 1994).

Recently the american Department of Defense proposed a system architecture to build and integrate parallel and distributed simulations, called HLA – *High-Level Architecture* (DMSO/DoD, 1997). This architecture defines three aspects of a distributed simulation: Federation rules, Object Model Templates, and Run-Time Infrastructure. As it looks to define relationship between model objects, model structures and system services interfaces, it constitutes a major change in the way simulations are built. This year the OMG delivered a *Request for Information* concerning the integration between HLA and CORBA (OMG, 1998). Our proposal differs from HLA/CORBA in the way that it allows the integration of existing simulations and libraries. The HLA/CORBA proposal defines a completely new environment, paradigm and system interfaces to build and integrate simulations.

We are currently working on a prototype of the architecture, using the C++ simulation libraries *C++Sim* and and *JavaSim* (Little, 1994). The CORBA functionalities are being provided by the *Chorus/COOL* environment (Chorus, 1996). This framework will allow us to seamlessly integrate simulations developed in C++ and Java.

References

Birman, K. P. and A., J. T. (1985). Replication and fault-tolerance in the Isis system. In *ACM SIGOPS*, volume 10, pages 79–86.

Birtwistle, G. M., Dahl, O. J., Myhrhaug, B., and Nygaard, K. (1973). *Simula Begin*. Chartwell-Bralt Ltd.

Chandy, K. M. and Misra, J. (1979). Distributed simulation: a case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452.

Chandy, K. M. and Misra, J. (1981). Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11):198–206.

Chorus (1996). *CHORUS/COOL-ORB r3 - Product Description*. Chorus Systems. CS/TR-95-157.3.

DMSO/DoD (1997). *The High Level Architecture*. http://hla.dmso.mil.

Fujimoto, R. M. (1988). Lookahead in parallel discrete event simulation. In *Proceedings of the 1988 International Conference on Parallel Processing, Pennsylvania*, pages 34–41.

Fujimoto, R. M. (1990). Parallel discrete event simulation. *Communications of the ACM*, 33(10):31–53.

Little, M. C. (1994). *C++SIM User's Guide*. University of Newcastle Upon Tyne, UK. Public release 1.5, draft version 1.0.

Lomow, G. and Baezner, D. (1991). A tutorial introduction to object-oriented simulation and Sim++. In *Proceedings of the 1991 Winter Simulation Conference*, pages 157–163.

Mallet, L. and Mussi, P. (1994). Object oriented parallel discrete event simulation: The PROSIT approach. Research report 2232, INRIA. Projet Mistral.

Martin, J. M. and Bagrodia, R. L. (1995). COMPOSE: An object-oriented environment for parallel discrete-event simulations. In *Proceedings of the 1995 Winter Simulation Conference.*

Mehl, H. (1991). Speed-up of conservative distributed discret event simulation methods by speculative computing. In *IEEE/ACM/SCS Workshop on parallel and distributed simulation*, Anaheim - California.

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

OMG (1995). *The Common Object Request Broker: Architecture and Specification*. Object Management Group. Revision 2.0.

OMG (1998). *OMG Request for Information on Distributed Simulation*. http://www.omg.org.

Raynal, M. (1992). *Syncronisation et état global dans les systèmes répartis*. Eyrolles. Collection EDF.

Righter, R. and Walrand, J. C. (1989). Distributed simulation of discrete event systems. *Proceedings of the IEEE*, 77(1):99–113.

Waldorf, J. and Bagrodia, R. L. (1994). A concurrent object oriented language for simulation. In *International Journal of Computer Simulation*, volume 4(2), pages 235–257.

C. A. Maziero *held a PhD. in Distributed Computing from the IRISA/INRIA - France, in 1994, and a MSc. in Industrial Automation from UFSC - Brazil, in 1990. Currently he works as associate professor and researcher in the Pos-Graduation Program on Applied Informatics of the Catholic University of Paraná State, in Brazil. His main interest areas are distributed systems, discrete-event simulations, and object systems.*