Using Computational Reflection in Optimistic Distributed Simulations

Fernando Vardânega, Carlos Maziero

Programa de Pós-Graduação em Informática Aplicada
Pontifícia Universidade Católica do Paraná
80.215-901 Curitiba - Brazil
e-mail: {*vardanega,maziero*}*@ppgia.pucpr.br*

Abstract

The recent studies in the distributed simulation area are focused in the High Level Architecture*, defined by the DoD/USA, which proposes a standard environment to develop and run distributed simulations. The HLA components are designed to ensure a high level of interoperability among simulations and, also, to allow maximum component reusability. This paper proposes a new mechanism to help optimist federates perform rollback procedures when needed. This mechanism uses computational reflection techniques to create a rollback manager meta-object that extends the low-level services provided by HLA.*

Keywords*: distributed simulation, HLA, time management, computational reflection.*

1 Introduction

In the last years, several distributed simulation systems have been built, allowing the simulation of complex systems like war scenarios, traffic systems, and others. The research activities in distributed simulation can be classed in two main areas. The first one, called PADS (*Parallel and Distributed Simulation*), has its emphasis on how to achieve high performance in distributed simulations while insuring all the causality relations between events. The second area, called DIS (*Distributed Interactive Simulation*), looks for the development of highly interactive simulation environments, allowing remote users to interact in real-time. Most of the efforts in this area were done by the US *Department of Defense* - DoD, pushed by its needs in military and emergency training [11].

Both research activities succeeded to achieve important goals in their areas. However, several problems remained to be solved, mainly related to the performance aspects, efficient network usage, reusability of the simulation code, and interoperability in heterogeneous computing environments. The US DoD *High Level Architecture* initiative [6] intends to define and develop a standard software environment in which heterogeneous simulation entities can interact using standard interfaces (see section 3). However, as shown hereafter, some HLA services are very low-level and hard to use when building simulation models that use peculiar time synchronization schemes.

2 Parallel/distributed simulation

The simulation model considered in PADS is roughly composed by an event set, generated in the execution of the processes modeling the system behavior, and variables that represent its state [7]. The events are scheduled to occur at a given time in the simulation time, and are put in an event list ordered by their timestamps (lower timestamps first). The time reference used in the simulation is generally not related to the real physical time, but serves only as a common virtual time reference for all processes that model the system being simulated.

The simulation execution is controlled by a scheduler mechanism, which continuously takes the first event in the event list, advances the simulation time to the event's timestamp and executes it. The sequential structure of the scheduler mechanism insures that the causality constraints between events are respected, because all events are processed in their chronological order. The main problem in PADS is how to build efficient scheduling algorithms to be run in a distributed environment, allowing event processing to be done in parallel, and insuring all their causality constraints. Two main approaches were proposed to solve this problem: the pessimistic (conservative) approach and the optimistic approach.

2.1 The pessimistic approach

This synchronization strategy was first proposed by Chandy, Misra, and Bryant [2]. The basis of this approach is to avoid violating the causality constraints local to each process. The interactions (causality relations) between processes are modeled by time-stamped messages containing events. Considering that each message contains a time-stamped event, the strategy local to each process consists on ensuring that all local events older than the events received from other processes will be processed before them. This implies on blocking the local event processing until the causality conditions can be verified. If the causality constraints local to each process are respected, the whole simulation can be considered correct, concerning the causality aspects [14].

2.2 The optimistic approach

In the optimistic strategy, local and received events can be processed without worrying about local causality constraints. Using this, process blocking is avoided, as all available events can be processed, and the simulation time can progress. If a message containing an old event arrives at a process, it will be needed to undo (rollback) the local simulation back to the timestamp of the old event, and then re-do the local simulation considering it. Using this strategy, the events that do not violate causality are confirmed, and the others should be canceled using a rollback mechanism. The first research works on optimistic algorithms were carried by Jefferson and Sowizral [10]; they had proposed the *Time Warp* method [15].

3 High Level Architecture

In 1995, the US Department of Defense started to define and build a standard architecture for the modeling and simulation of complex systems. It is a high-level, object-oriented software architecture, designed to ease the interoperability among different models and to allow component reuse. This architecture, known as HLA - *High Level Architecture*, constitutes a common technical framework for modeling and execution of distributed simulations [6]. Its main components are the *Object Model Templates* [5], the *HLA Compliance Rules* [3], and the *Runtime Infrastructure* [4].

3.1 Object Model Templates

Each HLA simulation is defined by a federation, in which a group of federates interact exchanging data and events. The definition of exchanged data and events is done using the *Object Model Templates* – OMT, which allows describing the objects that constitute the federation, their attributes and relationships. Each federation should define a *Federation Object Model* – FOM. This object model describes all the shared information (objects, attributes, associations, and interactions) used in the federation. Beyond FOM, there is also another object model, called *Simulation Object Model* – SOM, which describes objects, attributes, and interactions in a given simulation that can be used externally in a federation.

3.2 Compliance Rules

The compliance rules define ten basic rules that should be respected by a simulation to it be considered as according the HLA specifications. These rules define the responsibility and relationships among the federation components, including the federation itself, its federates, and the RTI. Five of these rules are applicable to the federations and the other five are applicable to the federates [3].

3.3 Runtime Infrastructure

The federates interact using the *Run-Time Infrastructure* – RTI, which can be seen as a distributed generic operating system that provides communication and coordination to the federates. All the communication in the federation should be done through the RTI; using this, the federates can be located in any computer connected to the network.

The interaction between a federate and the RTI uses methods calls from two different classes: *RTIAmbassador* and *FederateAmbassador*. The *RTIAmbassador* class contains all methods offered by the RTI to the federates. Its implementation is done by the RTI and is not accessible to the simulation programmer. On the other hand, the *FederateAmbassador* class is an abstract class, implemented by the simulation programmer, that identifies all methods that each federate should provide to the RTI for callback operations on the federates.

The services provided by the HLA to federates are classed in six categories [6]: *Federation Management*, *Declaration Management*, *Object Management*, *Ownership Management*, *Time Management*, and *Data Distribution Management*. The focus of this paper is on time management. The services that are provided by this category aims to coordinate the logical time advance and its relationships with the physical time.

4 Time management

The time management services coordinate the evolution of the federation logical time. Each can use a different time policy, i.e. can have a specific behavior with respect to the federation logical time. A federate is using a time-regulating policy if it can interfere in the time evolution of other federates. These federates control the time advance of federates using a time-constrained policy, by sending them messages associated to dates in the federation time. Thus, a federate can be regulating, constrained, regulating and constrained, or not regulating nor constrained. Initially all federates are neither regulating nor constrained; the shift for other time policy should be done by calling RTI methods. In a given federation, it is possible to have federates using any of these time policies.

The time management inside HLA is made up by two components that should be presented in more detail: *message ordering* and *logical time advance*.

4.1 Message ordering

Much of the time management is done by the correct ordering of messages coming from the federates and stored in the RTI. The messages are queued according the existence of timestamps (TSO - *Timestamp Ordered messages*) or not (RO - *Receive Ordered messages*), and according the time policies used by the sender and the receiver. Received RO messages are simply put in the FIFO input queue of the receiving federate, and are immediately available to the federate. On the other hand, received TSO messages are time-stamped with their sending times, and are put in the time-ordered queue of the receiving federate, and delivered to the federate in a non-decreasing timestamp order. A TSO message is delivered to the federate only when the RTI can insure that no more messages having a smaller timestamp will be received by that federate.

4.2 Logical time advance

The logical time advance in the federates is done explicitly, that is, the federate requests the RTI to advance its logical time and then waits for the confirmation of that request. This procedure is needed to insure that the federate will not receive any TSO message with a timestamp smaller than its local logical time. This condition can be guaranteed by the TSO message delivering mechanism of the RTI. Thus, the federate logical time only can advance when authorized by the RTI.

4.3 Time advance mechanisms

In the HLA, the time management of a federate is done in three steps, involving the federate (methods from *RTIAmbassador* and *FederateAmbassador* classes) and the RTI. The *RTIAmbassador* class is defined in the *Local RTI Component* – LRC, which contains all the methods that can be called by the federate on the RTI. The *FederateAmbassador* class is implemented by the federate code; it contains all the methods called by the RTI on the federate (callbacks). The figure 1 shows these interactions.

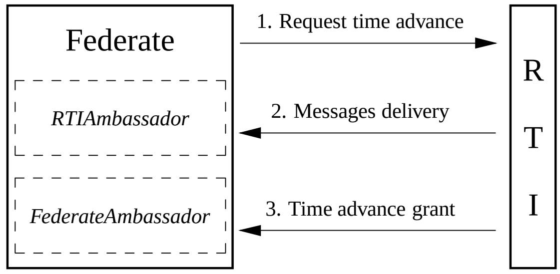

Figure 1. Time advance interactions

Due to the large diversity of simulations, the requirements in time management can vary largely from a simulation to another. The three most common approaches for time management in HLA are *time stepped*, *event driven* (pessimistic) and *optimistic* [8].

In the case of the pessimistic approach, the events should be processed according to the order of their timestamps, thus the logical time advance is bound to the events timestamps. This approach corresponds to the event-driven mechanism in HLA. In the optimistic approach, the events can be processed out of timestamp order. The RTI offers services for message delivering without considering timestamps of TSO messages, and basic rollback mechanisms. However, the rollback mechanisms provided by the RTI cover only the RTI state recovery (message queues, etc). All the management for state saving and recovery in the federate itself should be implemented by the simulation programmer.

Our work, presented in this paper, consists in the use of computational reflection techniques to build a *rollback manager*. This meta-object is charged to detect causality violations and to provide all state saving and rollback mechanisms needed by the federate, in a transparent way.

5 The rollback manager

Using the optimistic approach, the messages carrying events are given by the RTI to the federates without considering their timestamp order. If the federate receives a message in its past, there are basic mechanisms to recover from the causality error. Our proposal uses techniques of computational reflection [13] to create a rollback manager meta-object. This manager is charged of all the state recovery operations (rollback operations) on the optimistic federates, in order to improve the basic services offered by the RTI and also to turn easier the development of optimistic federates.

5.1 Computational reflection

Computational reflection is a development technique that allows a system to interact with itself, through a self-representation. Using this, the system can control its own behavior, allowing a clear separation between the functionality provided by the system to end users and the functions provided to configure and manage the system. This is done through a set of structures used by the system to represent its own aspects, both structural and computational [13].

According [13], a reflexive architecture computational system is constituted by two levels: a *base level* and a *meta level*. The base level is responsible for solving problems belonging to an external domain, normally related to the system's functionality. The meta level is in charge of the control and management of the base level. This allows a better modularity, separating the application code (base level) from the management code (meta level).

These concepts can be easily applied to object-oriented systems, associating a *meta-object* to each system object, now called a *base object*. This meta-level structure does not need to be applied on every system objects, but only to those which need to be managed or controlled.

The meta-object is causally connected to the base object, then any changes in the meta-object are be reflected in the base object. When an object is reflected, all its methods are also reflected on the correspondent meta-object. Thus, an invocation on a method in the basis object will be deviated to the corresponding meta-object method. The meta-object methods do their normal processing and eventually call their corresponding methods in the base object (figure 2).

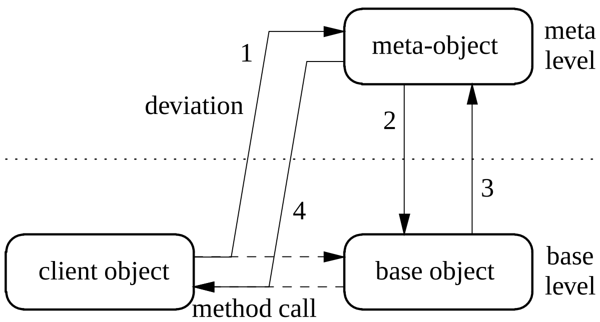

Figure 2. Computational reflection

5.2 The RTI rollback support

In order to receive all the messages without considering their time stamp order, the federate uses the method *flushQueueRequest* of the class *RTIAmbassador*. This service forces the RTI to give all the messages available in its internal queues to the federate. After this, the RTI invokes the callback method *timeAdvanceGrant* of the *FederateAmbassador* class, authorizing the federate's logical time to progress. If the federate receives a message with a time stamp smaller than some message already sent to it, some procedures should be executed to cancel this message and all the others received and processed after it. This recovery procedure is called *rollback*, and it includes unrolling the simulation to a execution point before the wrong message's timestamp. This procedure can imply in events reprocessing, canceling scheduled events, and canceling messages erroneously sent to other federates. The message cancellation is done using the RTI method *Retract*, used with the *flushQueueRequest* service, as shown in the figure 3.

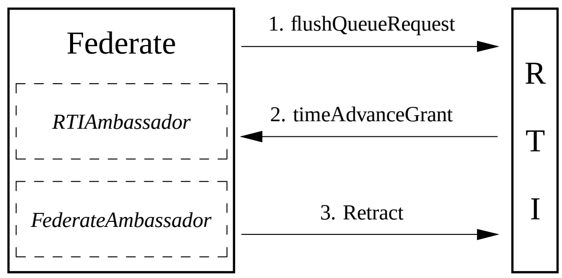

Figure 3. Optimistic federate retraction

If the message to be canceled is in the TSO queue of the receiver federate, it is simply removed from it by the RTI. However, if the message was delivered to the federate, its execution should also be rolled back, to cancel that message delivery. For doing this, the RTI calls the *requestRetraction* method on that federate, defined in its *FederateAmbassador* class. The federate should then undo processing done for events received improperly. If necessary, it should also use the RTI method *Retract* to cancel messages erroneously sent. All these actions should be implemented by the simulation programmer in the *requestRetraction* callback.

5.3 Automatic rollback management

The mechanism proposed here provides an automatic and generic way to deal with the *requestRetraction* callbacks, freeing the optimistic federates (and the programmer) of this complex task. Our proposal uses some computational reflection techniques [13] to create a time management meta-level between the RTI and each federate. The time management method calls between them are intercepted (reflected) by the rollback manager, which implements the rollback management in behalf of the federate. The figure 4 illustrates the general structure of the proposed mechanism:

Using this approach, the rollback manager takes to itself the control of the federate's state rollback, including

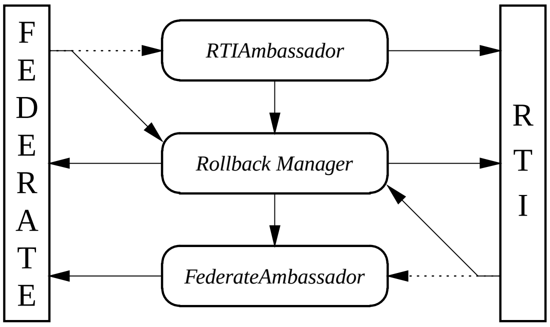

Figure 4. The rollback manager

canceling received or sent messages. The federate will continue calling the same methods of the *RTIAmbassador* class to interact with the RTI and it will receive RTI callbacks through the same *FederateAmbassador* class methods. However, some time management method calls will be intercepted and addressed to the rollback manager. Only some time management methods, mostly related to retraction operations, are intercepted; all the other methods are passed directly to the *RTIAmbassador* and *FederateAmbassador* implementations. The methods that should be reflected are those related to time management operations in optimistic federates, as *flushQueueRequest* and *timeAdvanceGrant*. Using this approach, the federate can adopt and optimistic behavior without worrying about possible rollbacks.

To the rollback manager be able to control rollbacks transparently, it should keep periodic snapshots of the federate's internal state (state checkpoints), in order to restore some previous state when a rollback occurs. For doing this, the rollback manager should have access to the federate's state at any time. However, generally the manager has no direct access to the federate's internal state. To overcome this, each federate should implement two callback methods that give controlled access to its internal state. As the rollback manager only needs access to the federate's state to save its current state and to restore a previous state, it is enough to implement two methods providing these operations[1]: a *getState*(S_c) method, which returns the federate's current state in the S_c state vector, and the *setState*(S_s), which restores the federate's state to the state saved in the state vector S_s. The rollback manager uses the *getState* method to maintain a list of previous states of the federate, and the *setState* method to restore a previous state, when a rollback occurs.

Using this approach, the implementation of optimistic federates becomes easier; its sole responsibility about rollbacks is the correct implementation of the methods *getState* and *setState* (figure 5).

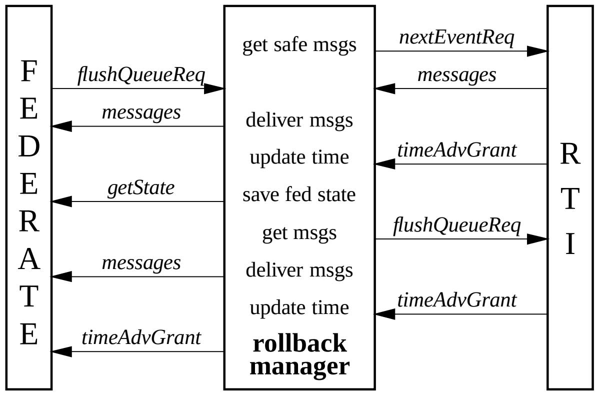

Figure 5. The proposed mechanism

Using the state saving methods, the rollback manager can save the federate's state at given times in which all TSO events sent to the federate are guaranteed. An event is considered guaranteed if it can be processed without any cancellation risk in the future, unless its retraction is explicitly requested[2].

The calls to the *flushQueueRequest* method are intercepted by the rollback manager, which interacts whit the RTI to obtain the TSO messages. This is done in two phases: initially the rollback manager uses a pessimistic approach to receive the TSO messages from the RTI. Through the method *nextEventRequest*, it requests that RTI deliver all the messages RO available inside its queue FIFO and all the messages TSO with time stamps smaller than the federate's current time. When there are not more TSO messages that match this requirement, the RTI authorizes the federate's time advance, through a callback to the *timeAdvanceGrant* method. This callback passes a future time value t_f as a parameter, to indicate that the federate's logical time can be advanced to t_f.

At this point, the manager had received all the safe messages as stated in the pessimistic approach (section 2.1), so RTI can guarantee that all the TSO messages with time stamps smaller than t_f had been delivered. This time t_f can be considered as a checkpoint time, indicating a point in the simulation time where the state of the federate is safe, with no rollback risks. Thus, the manager saves the federate's state at t_f as a checkpoint, using the *getState* call defined above.

After this pessimist phase, the manager calls the *flushQueueRequest* method on the RTI. At this point, the

[1]These two operations are inspired from the Isis system [1], for the replica's state management in groups of fault tolerant processes.

[2]The retraction of events is used inside of some discrete event simulations to model behaviors of interruptions and preemption, and that owe therefore to be directly requested by the federate.

RTI will deliver all other TSO messages sent to the federate, without worrying about their timestamps. These messages are considered unsafe and can suffer rollback, since the RTI doesn't guarantee that messages with smaller timestamps won't be sent to that federate in the future. If a rollback occurs, the rollback manager has access to all the needed information to undo the processing improperly done, to cancel scheduled events and to restore the last safe state of the federate.

5.4 The rollback procedure

If the federate receives a message older than its current logical time t_c, the federate's state should be rolled back to a previous safe state, in order to guarantee the causality constraints. The need of a rollback operation can be be detected by the rollback manager, because it receives all the messages addressed to the optimistic federate.

In HLA, there are four major event types that can change objects and their attributes. These events should be managed separately by the rollback manager, to allow it to maintain the whole control on all modifications performed in the federate. These events will be described in the next items of this text; at this point we can consider all the received events in a generic way.

When receiving a TSO message the manager will compare its timestamp t_m with the current logical time t_c (the rollback manager is at the same simulation time as the federate it manages). If $t_m < t_c$ a causality violation is detected, and the manager should restore the federate's state to a previous safe state $[S_s, t_s]$ with $t_s < t_m$.

The rollback manager should also keep track of all messages sent by the federate during $t_s < t \leq t_c$, i.e. after the S_s checkpoint, to be able to cancel them. Therefore, the manager can invoke the *Retract* method on the RTI to cancel messages sent to other federates. For doing this, the manager should keep track of all the message handles (*EventRetractionHandles*) for the messages sent during the time interval $[t_s, t_c]$.

In the same way, the manager can receive cancellation requests for messages improperly sent by other federates. The RTI will forward to the receiver the cancellation requests through the *requestRetraction* callback. Normally it is up to the federate to implement the needed procedures to deal with these cancellation messages. In our proposal, the rollback manager will take in charge this task.

5.5 The rollback manager operation

In our schema, the messages received are passed to the rollback manager and later forwarded to the optimistic federate. The messages received with attributes (*Attribute Handle Value Pair Set*) or parameters (*Parameter Handle Value Pair Set*) can be two: *ReflectAttributeValues* (RAV) and *ReceiveInteraction* (RI). In this case, before forwarding the messages to the federate, the rollback manager should save the old attribute values and the federate state to allow a possible rollback.

The mechanism operation can be summarized through the following algorithm, in which `F` stands for the federate, `M` for the rollback manager and `R` for the RTI:

while ($t_c < t_{max}$) do
- F: *next_event_time* = next local event time stamp ;
  flushQueueRequest(*next_event_time*) ;
- M: intercept the *flushQueueRequest* call ;
  nextEventRequest (*next_event_time*) ;
- R: send RAV/RI events to the federate ;
- M: intercept the RAV/RI events ;
- R: *timeAdvanceGrant* ;
- M: advance its logical time ;
  save the federate's current state ;
  forward the RAV/RI events to the federate ;
- F: receive the RAV/RI events ;
  store them in the queue of pending events ;
- M: *flushQueueRequest* (*next_event_time*) ;
- R: send remaining (unsafe) RAV/RI events ;
- M: intercept the unsafe events ;
- M: save attributes and recovery handles ;
  rollback the federate if needed ;
- R: *timeAdvanceGrant* ;
- M: advance the logical time ;
  forward the unsafe events to the federate ;
- F: receive the RAV/RI events ;
  store them in the queue of pending events ;
- M: *timeAdvanceGrant* ;
- F: advance the logical time ;
  process the events present in the queue of pending events ;

end

This mechanism can also be presented through a time diagram with all the interactions between the entities (federate, manager and RTI), as shown in the figure 6. This figure shows the interactions during the normal execution of an optimistic federate. When the manager detects a message older than the current time ($t_m < t_c$), it interacts with the federate and the RTI to execute the rollback.

The RTI normally calls the *requestRetraction* method on the federate when a message already delivered to it should be canceled. The event handler *EventRetractionHandle* for that message is passed with the request, which is intercepted by the manager. Using this handler, the rollback manager can recover the old values for the attributes or parameters. The old values were passed to the federate through the methods *ReflectAttributeValues* and *ReceiveIn-*

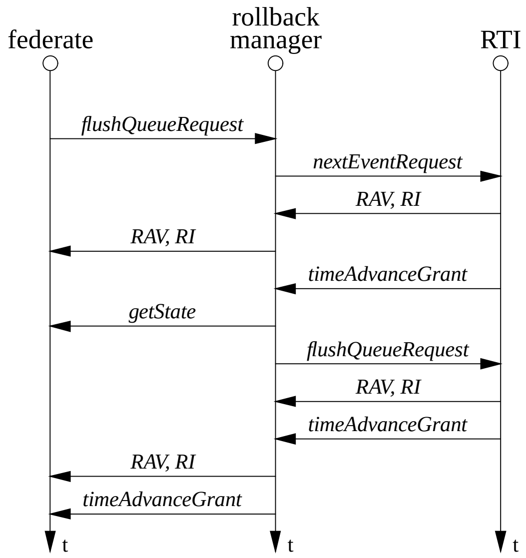

Figure 6. Time diagram for the interactions

teraction. Therefore, the federate doesn't need to worry about the cancellation of this event. If an improper processing has resulted in sending some messages to other federates, the manager will request their cancellation through the RTI method *Retract*. The rollback manager keeps track of all messages sent and their handles. The figure 7 shows the time diagram for this scenario:

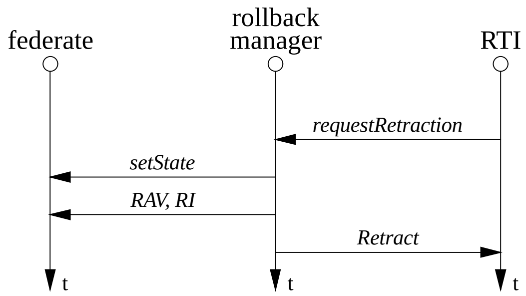

Figure 7. Time diagram for *requestRetraction*

6 Conclusion

The use of computational reflection techniques in the presented work showed to be useful, to simplify building optimistic federates. All the aspects related to rollback operations can be taken in charge by the rollback manager in behalf of the federate. This approach helps hiding the complexity of the optimistic approach from the simulation model programmer. The manager is capable to identify the need for a rollback, as well as to take all the proper actions to ensure that the federate returns to a safe state before the causality violation. It also takes for itself the responsibility of canceling messages improperly sent to other federates.

The rollback manager will accomplish tasks that are common to every optimistic federate, and does not depend on a specific federate behavior. The federate code becomes simpler, because the whole control and management of the rollback are under the manager's responsibility. All the optimistic federate should have is the correct implementation of the *getState* and *setState* operations.

To validate this proposal, a simple federation is being developed, involving several optimistic federates. In this work we are using RTI version 1.3 for C++. A Java binding package is also being used for the development of Java federates. Our development platform is a Solaris 2.6 Sun workstation with the Java Development Kit 1.1.6. The prototype federation is currently being built, and some preliminary measurements are being done. They show that the impact of the proposed mechanism on the system performance remains acceptable, but more extensive measurements should be done before giving concrete results.

The tests carried out with the rollback manager presented in this paper were done by manually substituting the RTI method calls, to the meta-object methods. This procedure was used for the validation of the proposed mechanism. With the use of a reflective language, the method deviations can be done in a transparent way. Such a language allows to define and transparently manage reflective objects. All the method invocations to the base objects are transparently deviated to their respective meta-level objects.

There are several programming languages supporting meta-object protocols [12]. One of them can be used to implement the rollback manager proposed here. The most used languages are: *CLOS*, *OpenC++* and *OpenJava*. As all the RTI code is available in C++, the OpenC++ language would be a good choice, as it uses the C++ syntax. In the specific case of our proposal, a better choice would be OpenJava [16].

In OpenJava, all the reflective objects are defined through the *OpenJava MOP* (Meta-Object Protocol). The OpenJava code is pre-processed to generate standard Java code. However, OpenJava is not yet mature (current version is 1.0) and does not support some characteristics essential to the development of distributed simulations using the HLA architecture. As OpenJava, there are other proposals for Java reflective implementations, like MetaXa [9], that could be incorporated to this work.

References

[1] K. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, December 1993.

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

[3] Defense Modeling and Simulation Office - US DoD. *HLA Compliance Rules*, 1996. http://www.dmso.mil/dmso/docslib/.

[4] Defense Modeling and Simulation Office - US DoD. *HLA Interface Specification*, 1996. http://www.dmso.mil/dmso/docslib/.

[5] Defense Modeling and Simulation Office - US DoD. *HLA Object Model Templates*, 1996. http://www.dmso.mil/dmso/docslib/.

[6] Defense Modeling and Simulation Office - US DoD. *HLA Overview*, 1997. http://www.dmso.mil/dmso/docslib/.

[7] A. Ferscha and S. K. Tripathi. Parallel and distributed simulation of discrete event systems. Technical report, University of Maryland, August 1994.

[8] R. Fujimoto. Time management in the high level architecture. *SCS Simulation Magazine*, December 1998.

[9] M. Golm. Metaxa and the future of reflection. In *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, October 1998.

[10] D. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[11] Kanarick. A technical overview and history of the simnet project. *Advances in Parallel and Distributed Simulation, SCS Series*, 23, 1991.

[12] G. Kiczales, M. Ashley, L. Rodriguez, A. Vahdat, and D. Bobrow. Metaobject protocols: Why we want them and what else they can do. *Object Oriented Programming: The CLOS Perspective*, 1993.

[13] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 147–156, october 1987.

[14] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(4):39–65, March 1986.

[15] H. Sowizral and D. Jefferson. Fast concurrent simulation using the time warp mechanism. In *Distributed Simulation, SCS, Simulation Councils*, pages 63–69, La Jolla, California, 1985.

[16] M. Tatsubori. *OpenJava Tutorial*. Tsukuba University - Japan, 1997. http://www.softlab.is.tsukuba.ac.jp/∼mich/openjava.