A Generic Rollback Manager for Optimistic HLA Simulations

Fernando Vardânega, Carlos Maziero
Programa de Pós-Graduação em Informática Aplicada
Pontificia Universidade Católica do Paraná
80.215-901 Curitiba - Brazil
E-mail: { vardanega,maziero}@ppgia.pucpr.br

Abstract

This paper describes the addition of an extra piece of software, a rollback manager, to implement state saving and rollback management for optimistic federates in the High Level Architecture (HLA). This mechanism uses computational reflection techniques to create a rollback manager meta-object that extends the low-level time management services provided by HLA. The main propose of the rollback manager is to relieve the federate from the burden of handling problems related to the federate state saving management and recovery. Some experimental results are shown, to prove the feasibility of the proposed mechanism.

1. Introduction

The research activities in distributed simulation can be classed in two main areas. The PADS (*Parallel and Distributed Simulation*) area has its emphasis on how to achieve high performance in distributed simulations while insuring all the causality constraints between events being processed in parallel. Two main approaches were proposed to solve this problem: the conservative approach [1] and the optimistic one [2] [3]. 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.

Several problems remained open in both areas, mainly related to performance aspects, efficient network usage, simulation code reuse, and interoperability in heterogeneous environments. To cover these issues, the US DoD proposed the *High Level Architecture* initiative [4], defining a standard architecture for the modeling and simulation of complex systems. It is a software environment designed to ease the interoperability among different models, through standard interfaces. Also, it uses object orientation techniques to allow component reuse.

However, as shown hereafter, some HLA services are very low-level and hard to use when building simulation models that use peculiar time synchronization schema. In this paper we present a generic rollback manager, able to detect causality violations and providing all the state saving and rollback mechanisms needed by optimistic simulation entities, transparently.

This paper is organized as follows: section 2 presents the main HLA architectural aspects; section 3 explores the HLA time management mechanisms; section 4 presents the computational reflection concepts used to define the rollback manager, which is fully defined in section 5; finally, section 6 shows some experimental results.

2. The High Level Architecture

The *High Level Architecture* constitutes a common technical framework for modeling and execution of distributed simulations. Its main components are the *Object Model Templates*, the *HLA Compliance Rules*, and the *Runtime Infrastructure* [4].

Each HLA simulation is defined by a *federation*, in which a group of *federates* interact exchanging data and events. These interactions are defined using the *Object Model Templates* - OMT, which allows describing the objects that constitute the federation, their attributes, and relationships. Each federation defines a *Federation Object Model* – FOM, describing all the shared information (objects, attributes, associations, and interactions) used in the federation.

Beyond the FOM, the *Simulation Object Model* (SOM) describes objects, attributes, and interactions that can be used externally. To be considered as according the HLA specifications, the federation should respect the ten *HLA Compliance Rules*. They define the responsibility and relationship among all the federation

The federates interact using the *Run-Time Infrastructure* - RTI, which can be seen as a distributed

generic operating system that provides communication and coordination services to the federates.

All interactions in the federation should be done through the RTI. The interaction between a federate and the RTI uses method 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 federate itself.

The services provided by the HLA to federates are classed in six categories [4]. The focus of this paper is on the *Time Management* category, which provides coordination and logical time services to the federates.

3. Time Management in HLA

The main time management aspects covered by the HLA specification are the *federates' time policies*, the *message ordering definitions*, and the *logical time advance strategies*.

3.1. Time Policies

In HLA, federates can adopt different *time policies*, resulting in different behaviors with respect to the federation logical time. A federate can adopt a *time-regulating* policy, allowing it to produce time-stamped events. Some federates can use a *time-constrained* policy, forcing it to consume time-stamped events (sent by time-regulating federates).

Thus, a given federate can be *regulating*, *constrained*, *regulating and constrained*, or *not regulating nor constrained* (the initial default behavior).

The federates can enable or disable these time policies at any time, through RTI method calls. A federation can have federates using any of these time policies.

3.2. Message Ordering

Much of the time management is done by the correct ordering of messages sent by the federates. The RTI manages input queues for each federate. Messages are stored in the RTI queues according the existence of timestamps (TSO - *Time-Stamp Ordered* messages) or not (RO - *Receive Ordered* messages), and according the time policies used by the sender and the receiver.

RO messages are simply put in the FIFO input queue of the receiving federate, and are immediately available to it.

TSO messages are put in the time-ordered input queue of the receiving federate, and delivered to it in time-stamp order. A TSO message can be delivered to a federate only when no more messages having a smaller timestamp will be received by that federate.

3.3. Time Advance Approaches

The logical time advance in the federates is done explicitly: the federate requests the RTI to advance its logical time and then waits for a confirmation callback. 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 should be guaranteed by the TSO message delivery mechanism of the RTI. Thus, the federate logical time only can advance when authorized by the RTI.

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* and *optimistic* [5]. In the event-driven approach, the events are processed according their timestamp order, thus the logical time advance is bound to the events timestamps. This corresponds to the conservative approach. In the optimistic approach, the events can be processed in any order.

3.4. The Optimistic Approach

In the optimistic approach, the messages carrying events are delivered to the federates without considering their timestamp order.

The federate uses the *flushQueueRequest* method to ask the RTI for all TSO messages present in its input queue. After delivering the messages, the RTI invokes the callback method *timeAdvanceGrant* in the federate, authorizing it to advance its logical time.

If the federate receives an out-of-order time-stamped message, it should rollback its local execution, to correctly consider the ordering of all messages received. This recovery procedure includes unrolling the simulation to a execution point before the out-of-order message timestamp, re-processing events, canceling scheduled events, and canceling messages erroneously sent to other federates.

The message cancellation is done using through RTI method *Retract*, used with the *flushQueueRequest* service (figure 1).

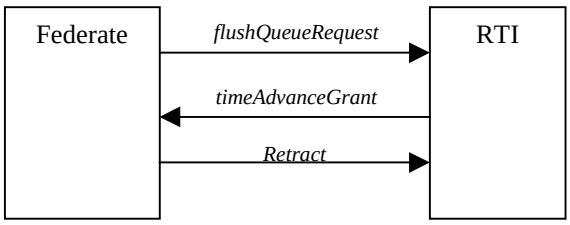

Figure 1. Optimistic federate retraction

If the message to be canceled was already delivered to another federate, it should also be rolled back. The RTI calls its *requestRetraction* callback method, and the federate should then undo any processing done for events received improperly. All these federate actions should be implemented by the simulation programmer.

Our work, presented in this paper, consists in the use of computational reflection techniques to build a generic rollback manager. This manager is charged to detect causality violations and to provide all state saving and rollback mechanisms needed by a federate, in a transparent way. It frees the simulation programmer of programming tasks not related to the simulation model itself.

4. 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. According to [6], 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).

5. A Rollback Manager

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 [6] 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 2 illustrates the general structure of the proposed mechanism:

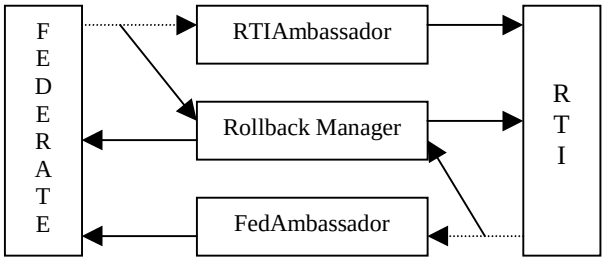

Figure 2. The rollback manager

Using this approach, the rollback manager takes to itself the control of the federate's state rollback, including 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 an optimistic behavior without worrying about possible rollbacks. For 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. Thus, the rollback manager should have access to the federate's state at any time. To provide that, 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, t_c) method, which returns the federate's current state in the [S_c, t_c] state vector, and the *setState*(S_c, t_c), which restores the federate's state to the state saved in the state vector [S_c, t_c]. 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

[1] These operations were inspired from Isis System [7], for the replica´s state management in groups of fault tolerant processes.

the correct implementation of the methods *getState()* and *setState()*.

Using the state saving methods, the rollback manager can save the federate state at given times in which all TSO events sent to the federate are guaranteed. An event is considered safe if it can be processed without any cancellation risk in the future, unless its retraction is explicitly requested. The calls to the *flushQueueRequest* method are intercepted by the rollback manager, which interacts with the RTI to obtain the TSO messages.

This is done in two phases: initially the rollback manager uses a conservative 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 input queue and all the messages TSO with time stamps smaller than the federate current time. When there are no more TSO messages that match this requirement, the RTI authorizes the federate time to 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 pessimistically received all the safe messages, so the 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 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.

5.1. 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 rollback manager can detect the need of a rollback operation, 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 state. 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 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 $t_s < t < t_c$, i.e. after the $[S_s , t_s]$ 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 $[t_s ; t_c]$. Also, 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 perform this task.

5.2. 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.

This mechanism can be presented through a time diagram with all the interactions between the entities (federate, manager and RTI), as shown in the figure 3. 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 *ReceiveInteraction*.

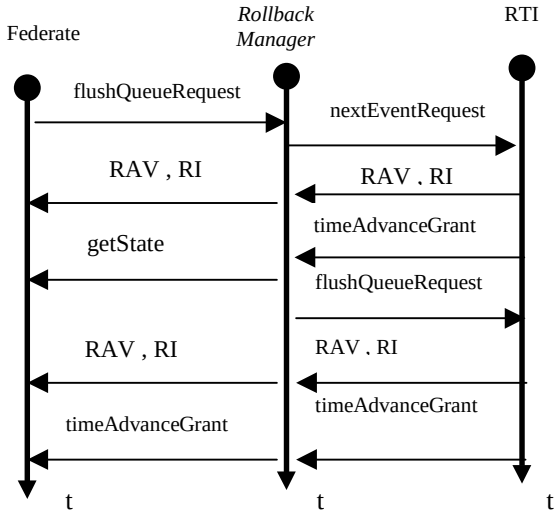

Figure 3. Time diagram for the interactions

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.

6. Experimental Results

To validate this proposal, a simple federation involving several optimistic federates was developed. In this work we are using RTI 1.3 version 6 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 tested, and some measurements are done. In this simulation, the federate state size was 200 bytes.

The inclusion of a rollback manager increased the total execution time of this simulation. This overhead is due to the reflection on method invocations. These method invocations represent the message transfers among federates and the RTI. The message timestamps are checked by the rollback manager, in order to the causality constraints. This procedure adds some overhead to the overall simulation performance. In order to measure this overhead we did some experiments with the prototype simulation. The first experiment was built changing the number of interactions in the simulation execution. First, the original was executed several times to obtain the average execution time. The number of interactions ranged from 500 to 2000 to measure the simulation performance without the rollback manager. The values, in m:ss format, obtained are shown in figure 4.

Simulation	Number of Interactions			
	500	1000	1500	2000
1	2:21	4:25	6:42	8:30
2	2:42	4:24	6:44	8:16
3	2:29	4:35	6:12	8:18
4	2:33	4:31	6:37	8:22
5	2:41	4:22	6:23	8:27
Average:	2:33	4:27	6:32	8:23

Figure 4. Execution times without the Rollback Manager

The same experiment was repeated adding the rollback manager to perform the rollback procedure, instead of leaving this task to the federate. The execution times obtained showed us the overhead caused by the inclusion of the rollback manager. The values, in m:ss format, are shown in the figure 5.

Simulation	Number of Interactions			
	500	1000	1500	2000
1	3:34	6:21	9:16	12:13
2	3:36	6:14	9:34	12:16
3	3:44	6:22	9:22	12:23
4	3:22	6:31	9:45	12:12
5	3:52	6:33	9:39	12:17
Average:	3:38	6:24	9:31	12:16

Figure 5. Execution times with the Rollback Manager

In this simulation, two messages are exchanged among the federates through the RTI. These messages must be intercepted and analyzed by the rollback manager, so this procedure adds an extra processing time to the execution. Therefore, when the number of messages increases, the computational overhead increases as well. The figure 6 shows the execution times before and after adding the rollback manager.

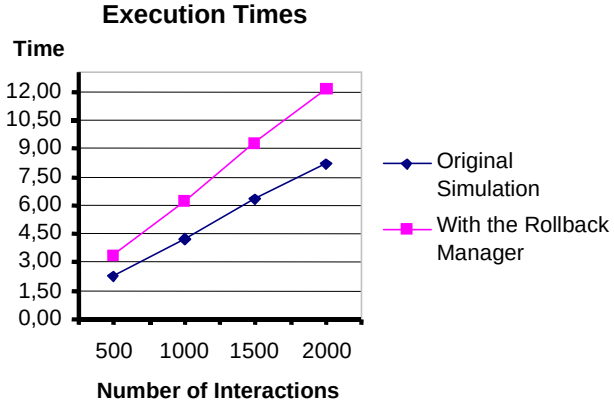

Figure 6. Difference among execution times (in minutes)

One important point to be analyzed is the relationship among the execution times. As suggested in [8], this relationship defines the overhead factor by computational reflection techniques.

The overhead factors are calculate dividing the execution time with the rollback manager by the original execution time. They are shown in the figure 7.

	Execution Times			
Number of Interactions:	**500**	**1000**	**1500**	**2000**
Without Rollback Manager	2:33	4:27	6:32	8:22
With Rollback Manager	3:38	6:24	9:31	12:16
Factor	1.45	1.46	1.47	1.48

Figure 7. Overhead Factors

The results of these measurements show that the simulation execution with rollback manager is almost 1.5 times slower than the original simulation execution. The evolution of this overhead factor is presented on figure 8.

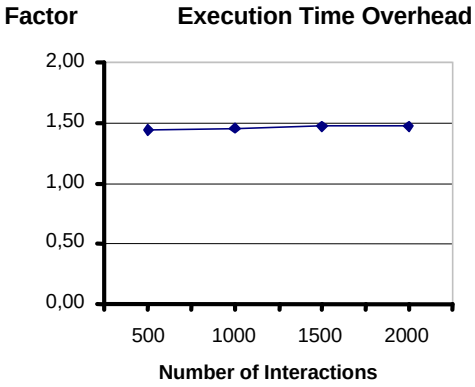

Figure 8. Overhead Factors Evolution

If the increased overhead factor of computational reflection techniques is under a 10 factor, the advantage of those techniques is worthwhile [8]. The results of these measurements show that the factor is almost stable in 1.5, so the use of a reflexive rollback manager can be considered worthwhile.

This overhead can be reduced by a proper application design. The simulation prototype was design just to show the proposed mechanism's feasibility. The source code could be optimized to reduce this overhead.

Another very important point in this proposed mechanism is the fact that the rollback manager is initialized only when the federate assumes a optimistic behavior. Thus, if the federate never invokes the *flushQueueRequest* method from the *RTIAmbassador* class, the simulation performance will not be affected.

The rollback manager was implemented as an additional class that is instantiated only when the federate invokes the *flushQueueRequest* method, as mentioned before. When this class is instantiated, there is an increase of the total memory amount allocated by the simulation execution. This memory utilization increase was measured and the values are showed in the figure 9.

Simulation	**Allocated Memory (Mb)**	
	Total	**Resident**
Without RM	15	10
With RM	16	13
Overhead	6,67%	30%

Figure 9 : Memory Allocation

The memory utilization was calculated using standard Solaris operating system commands and tools. Thus, these commands outputs are not exacts but can be used to measure the memory allocation overhead.

The impact of the proposed mechanism on the system performance remains acceptable, but more extensive measurements should be done before giving concrete results.

7. Conclusions

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.

The tests carried out with the rollback manager presented in this paper were done by manually substituting the RTI method call, 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 [9]. 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 [10]. 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 [11], that could be incorporated to this work.

8. References

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

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

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

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

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

[6] Maes, P., "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.

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

[8] Chiba, S. and Masuda, T., "Designing an Extensible Distributed Language with Meta-level Architecture", *Proceedings of ECOOP`93*, pags 482-501, Germany, 1993

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

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

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