Proceedings of the 1987 Winter Simulation Conference
A. Thesen, H. Grant, W. David Kelton (eds.)

TIMELOCK: A CONCURRENT SIMULATION TECHNIQUE AND ITS DESCRIPTION IN SMALLTALK-80

Jean Bezivin

Summary:
As part of a general project to build an experimental evaluation environment for object-oriented simulation, several tools have been developed. This paper first presents a client-server approach to simulation within the framework of the Smalltalk-80[1] programming system. This basis is then extended to describe a distributed simulation scheme called *timeLock.*

keywords: Discrete-event simulation; Parallel simulation techniques; Object-Oriented languages; Smalltalk-80; timeLock.

1. Introduction.

The object-oriented programming paradigm is familiar to simulation programmers. It is therefore a natural trend to take advantage of modern object-oriented languages and systems to improve the production of simulation software. Following an approach similar to the DEMOS system built upon the Simula 67 language (Birtwistle 1979), we have defined an evaluation platform on top of the Smalltalk-80 programming system. Within the framework of this platform called SIMTALK, several aspects of the methodology for producing simulation software are being investigated: graphical programming, interactive programming, automatic tracing and statistics gathering mechanisms, advanced programming paradigms useful for simulation (e.g. constraint programming), etc. In the present paper we start from a client-server decomposition model and we first show how such a model can be expressed within the defined framework. We are then in a position to present the client-server concurrent simulation technique called timeLock and to describe its main characteristics.

TimeLock, an algorithm first presented in (Bézivin and Imbert 1982), is a concurrent simulation technique based on a client-server approach. The basic idea is to associate to every client (or process) a local virtual time and to allow it to execute its local actions without control. When a client requests a service from a server (or monitor), it is only allowed to go on if its local time is not greater than a global virtual time. If this is not the case, then the process is said to be anticipating and accessing the server is not allowed (the monitor is locked for it). The algorithm will be briefly presented and a description of it will be given in the Smalltalk language. TimeLock may be compared to another more innovative distributed simulation scheme called Time Warp (Jefferson 1985). On the contrary of timeLock, Time Warp does not follow a client-server but an actor network approach. One of the project goals is also to use an evaluation model of both techniques within the framework of Smalltalk-80 in order to compare their behaviors and performances.

[1] Smalltalk-80 is a trademark of Xerox Corporation

2. Basic simulation and synchronization scheme.

A simulation problem may be structured according to several schemes. One particular choice is between an actor network and a client-server decomposition. In the first case all the simulation entities are of similar nature and interact by sending messages to one another. In the second case, a simulation entity may be an active one (a client) or a passive one (a server). Clients communicate with servers by sending requests.

The client-server model seems to be a natural structuring way in Smalltalk-80 and we have based the main part of the present work on this scheme. Our starting point is a simplification of the model presented in (Goldberg and Robson 1983). Instead of using two classes Simulation and SimulationObject, we use only one class SimulationObject which defines all the basic simulation mechanisms. This class will have two subclasses respectively representing clients and servers:

- ActiveSimulationObject
- PassiveSimulationObject.

The class SimulationObject representing the self contained simulation kernel, will offer the following primitives:

holdFor: a Duration suspends a simulation object for a given amount of simulated time.
startProcess registers an activation or reactivation of a simulation object process.
stopProcess register a temporary or permanent deactivation of a simulation object process.
proceed executes one simulation step.
time returns the current value of the simulation time.
tasks is intended to be redefined by subclasses in order to specify the algorithmic behavior of various simulation objects.

In order to provide these services, the following class variables are used:

CurrentTime The current value of the simulation time.
ProcessCount The number of active simulation processes.
TimeQueue The queue of simulation processes blocked as a result of a *holdFor:* operation in increasing order of their reactivation time.

Very often cooperation and communication schemes for simulation processes are quite complex. The simulation programmer needs high level mechanisms to deal with these situations. In our client-server model usually the server control access to a given set of resources shared between a number of user processes. The programming paradigm corresponding most closely to a shared resource is the monitor concept (Hoare 1974). A monitor is basically a data structure accessible only by predefined entry points and protected in a mutual exclusion scheme. At any time only one process may be operating on the protected data of a given monitor. Moreover, a monitor may contain so called *condition variables* representing waiting queues of processes. A process may enter a waiting queue (*wait* operation) or may awake another process blocked on a waiting queue (*signal* operation).

Our first effort was thus to provide monitors and conditions within the framework of Smalltalk-80. Using the mechanism described in (Bézivin 1987) if x is a given object and if we evaluate the following expression:

y <- x enveloped

then object y is functionally equivalent to x with the added property of method encapsulation, i.e. any call to a method of y results in the corresponding call to a method of x bracketed by a previous call to the *prologue* method and a subsequent call to an *epilogue* method. The object x is supposed to answer both *prologue* and *epilogue* methods.

To give an implementation of monitors using the method defined above, we followed directly the translation scheme suggested in (Hoare1974). For every monitor, two semaphores were used: one for mutual exclusion and another for holding processes that lose control as a result of issuing a successful *signal* operation on a condition.

The Smalltalk simulation framework consists of a number of hierarchically related classes. A typical architecture of a simulation program is shown in figure 1. In this example several *ships* and *lorries* exchange goods through a *harbour*.

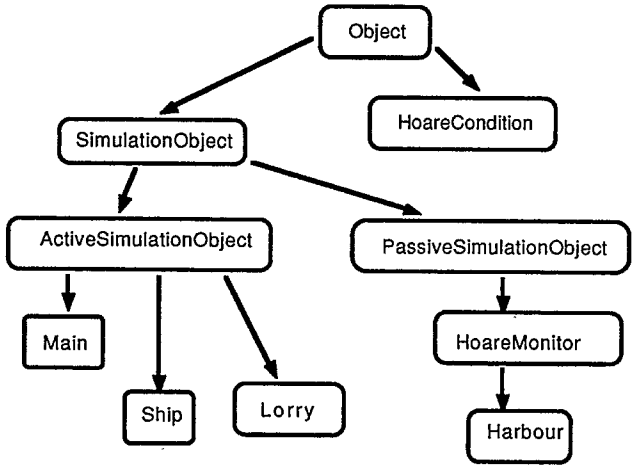

figure 1: an example showing related classes in a simulation program.

3. A distributed simulation approach.

All the classical discrete-event simulation techniques suffer from a common drawback: there is a severe limitation on the duration and size of such simulations when executed on single processor configurations. It is widely recognized (Misra 1986) that the only way out of this situation is to use concurrent discrete event-simulation techniques, for example on a network of processors.

A desired quality for a concurrent simulation technique is transparency. This means that ideally the expression of a simulation program should not have to be changed when converted from a non-distributed to a distributed environment. The approach to concurrent simulation that we are going to present has the transparency property. Starting from a client-server simulation model as the one outlined in figure 1, it is necessary to state on which site (i.e. a processor with its memory) of the network each entity will be installed. But the expression of the individual entities algorithms and of the interaction between these entities will not have to be modified whatsoever.

The presentation will be done in two stages. First the basic time anticipation scheme will be described. Then the multi-site underlying distributed configuration will be taken into account and a model for implementation will be given.

3.1. A simplified model.

The basic idea is to define a Global Virtual Time (GVT) common to all processes and a Local Virtual Time (LVT) for each simulation process. A process is said to be anticipating when its LVT is strictly greater than the GVT. It will then see all monitors as being virtually busy. The rules to make this scheme work are:
- the message *time* delivers the value of the LVT when the call is made inside a process and the value of the GVT when it is made inside a monitor.
- every process is allowed to execute local actions whatever its LVT and the GVT might be.
- when a process is anticipating, every monitor is virtually busy for it, until the GVT catches up with its LVT.

The basis of the Smalltalk implementation may be outlined as follows. First a variant of SimulationObject called DistributedSimulationObject defines the general simulation mechanisms. A variable *MonitorGate* is added to hold the simulation processes that access a monitor while they are in anticipation state. The method *proceed* is redefined to update the GVT whenever the number of simulation processes becomes zero. The new value for the GVT is then computed. All the processes blocked on *MonitorGate* with a LVT equal to the new value of the GVT are then freed. A partial description of this class is given below:

```
Object subclass: #DistributedSimulationObject
classVariableNames:
        'GVT ProcessCount MonitorGate'
instance methods:
holdFor:aDuration
        self subclassResponsibility
time
        self subclassResponsibility
class methods:
initialize
        GVT<-0.0.
        ProcessCount<-0.
        MonitorGate<-SortedCollection new
startProcess
        ProcessCount<-ProcessCount+1.
stopProcess
        ProcessCount<-ProcessCount-1
newProcessFor:aBlock
        self startProcess.
        [aBlock value. self stopProcess] fork
```

The DistributedSimulationObject class has now two subclasses named:
- ActiveDistributedSimulationObject
- PassiveDistributedSimulationObject

The main characteristics of these classes are defined below. First active objects will own an instance variable *localTime* and the operations *holdFor:* and *time* will get a very simple implementation with respect to this variable. A more subtle problem is to ensure that when a process starts another process, the starting time of the son process will be the local time of the father process when the operation is performed. This is defined by the *start* method. One exception is for the initial process of the simulation who will be the only one started by an external operator. The *coldStart* method handles this case.

```
DistributedSimulationObject subclass:
#ActiveDistributedSimulationObject
instanceVariableNames:'localTime'
instance methods:
holdFor:aDuration
        localTime<-localTime+aDuration
setTime:aTime
        localTime<-aTime
tasks
        self subclassResponsibility
time
        ^localTime
class methods:
start
|itsLocalTime|
itsLocalTime<-thisContext sender receiver time.
self newProcessFor:
        [(self new setTime:itsLocalTime) tasks]
coldStart
self newProcessFor:
        [(self new setTime:0.0)tasks]
```

Passive objects differently defines the *time* and *holdFor:* methods. It may be noticed that a process can also be started from a passive object. In that case the starting time will be the current value of the GVT. Finally we have to take into account that the simulation time may change while executing a method in a passive object. As a consequence, the local time of the calling active object has to be updated at the end of the call.

```
DistributedSimulationObject subclass:
#PassiveDistributedSimulationObject
instance methods:
holdFor:aDuration
        self delayUntil:GVT+aDuration
delayUntil:aTime
|e|
        e<-DelayedEvent onCondition:aTime.
        MonitorGate add:e.
        DistributedSimulationObject stopProcess.
        e pause.
        DistributedSimulationObject startProcess
prologue
        ^self
epilogue
        ^self
time
        ^GVT
prologue:aLocalTime
        aLocalTime>GVT
        ifTrue:[self delayUntil:aLocalTime].
        self prologue
epilogue:theSender
        self epilogue.
        theSender setTime:GVT
```

3.2. Multiprocessor implementation.

What has been described so far is only one part of the timeLock technique, namely the anticipation mechanism. The previous model is self-contained and can be used by itself to experiment with some issues in concurrent simulation. However it overlooks a number of important facts:
- in a distributed implementation there will be no more global time.
- the notion of a site must be explicitly taken into acount.
- the global variable GVT has be replaced by a set of site estimations of the GVT, one per site.

The second part of the timeLock description is an abstract specification of the functions to be implemented to make the system run on a distributed configuration. Although this specification must be mainly regarded as implementation guidelines, it may also be considered as an operational model. In this case a functional simulation of the distributed configuration running the concurrent simulation system may itself be easily derived from the following description.

First every simulation object (active or passive) is associated with a given execution site. This is represented by an instance variable *mySite* defined in class DistributedSimulationObject. Then three more classes are defined:

SMessage will represent actual messages sent to various simulation objects. Several instance variables are defined here, representing the various attributes of a message: *sender, receiver, methodSelector, parameters.*

SimulationKernel will define the characteristics of the simulation kernels implemented on each site. Such a kernel is itself a system process running concurrently with the various simulation processes. All kernels communicate together. Each one handles calls from its site processes to other sites monitors. It also receives calls from other site processes to its own site monitors. In this case it will create a delegated process on its site in order to handle locally the monitor call. In addition to the *mySite* instance variable, each kernel defines the *GVT* variable representing the local site estimation of the global virtual time.

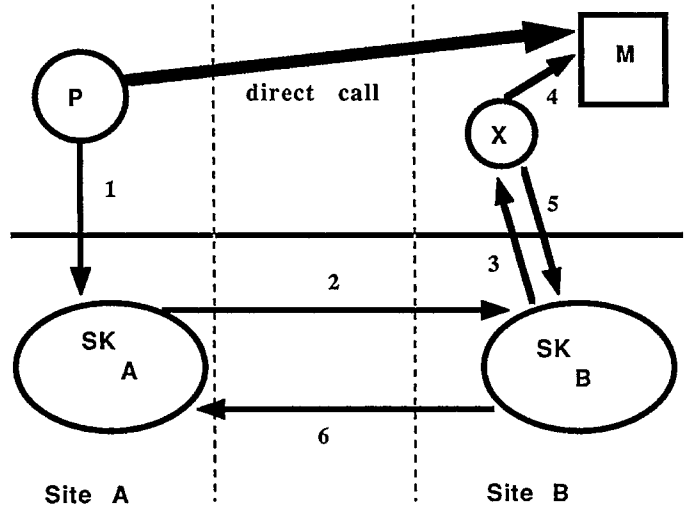

figure 2: inter-site call from a process to a monitor.

Delegator is the class corresponding to delegated processes that will be dynamically created to handle remote monitor call on the receiver site.

Let us suppose that process P, installed on site A, sends a request R to monitor M installed on site B, with A different from B. The transmission from object P to object M is first trapped and an instance Sm of class Smessage is created, recording in particular P, M and R. The direct call from P to M does not actually take place and is replaced by the following sequence of transmissions (figure 2):

1. Process P sends the message Sm to the simulation kernel of site a (SK_A). As a result process P gets blocked on a waiting queue of SK_A (the control is not returned to P).
2. SK_A delivers the message Sm to the simulation kernel of site B (SK_B).
3. SK_B creates a delegate process (X) that will request service R from monitor M on behalf of process P.
4. X sends message R to object M with the corresponding parameters.
5. When X receives control back from monitor M, it will send a message "end of mission" to SK_B with the relevant information.
6. SK_B informs SK_A that the call is finished and provides return information. As a result, process P is now allowed to go on.

On each site the simulation kernel is responsible for maintaining the local estimation GVT of the global virtual time. To this end it executes an iterative multi-state algorithm outlined below. An instance variable *myState*, defined in class SimulationKernel, indicates the current state of the site and takes in turn the values *normalState*, *counselState* and *estimationState*.

If *myState* = *normalWorking*, that means that there is at least one process running with its LVT equal to the GVT on the site.

If *myState* = *counselState*, that means that all processes on the site are either blocked or are anticipating. In this transient state, the site initiates a transaction with other sites in order to agree on a correct new value for the GVT. The first action is to signal to other sites that it is in *counselState* by broadcasting a message meaning: "I am ready to update the GVT".

If *myState* = *estimationState* that means that the site wants to update the GVT and has received counsel messages from all other sites. It computes the new value for the GVT for itself and broadcasts it to other sites. When it has itself received the estimations of all sites, the new value for the GVT is the minimum of all these values and its own one. After updating its GVT and releasing processes blocked in the local *MonitorGate* queue, the site state returns to *normalWorking*.

It is important to stress several facts about the algorithm:
a) The cost of this algorithm is basically two broadcast operations for each time update, one for the counsel messages and one for the time estimations.
b) If the distributed configuration has native broadcast facilities, they will be used for the implementation.
c) A usual simplification is to permit creation of simulation entities only on the site where their class is installed. This characteristic may be taken advantage of in an actual implementation.

Moreover this algorithm runs concurrently with the simulation processes installed on the same site and does not slow down significantly their performances.

4. Conclusion.

Using modern object-oriented languages such as Smalltalk-80 may bring a lot of benefits to the simulation programmer Increased quality of programs results from the enforced modularity. Development costs may be highly reduced due to the reusability property. At the origin of this reduction lies te powerful abstraction mechanism of class inheritance that was discovered by the designers of the Simula-67 language. The simulation framework that we have developed on top of the Smalltalk language follows an idea similar to DEMOS, a system for discete event modelling on Simula (Birtwistle 1979). However the Smalltalk basis seems to us much more flexible and powerful than the Simula basis.

Furthermore the facilities of Smalltalk enabled us to give a description of the concurrent technique timeLock, here again using the inheritance feature and a polymorphic style of programming.

The timeLock description in Smalltalk has the pleasant property of being operational, i.e. it is in fact a simulation of the distributed simulation mechanism itself. The first consequence is that it has been possible to check the transparency property of timeLock in the following way: a simulation program running within the conventional framework, for example the set of classes {Main, Ship, Lorry, Harbour} described in figure 1, works within the distributed framework without any modification. A second consequence that is being currently investigated is the possibility of using this framework to explore several open problems of distributed simulation. One of these problems for example is the need to present to the outside world a coherent view of what is going on inside the simulation. This is particularly difficult if the local virtual time of the various processes is different. This problem is perhaps one of the most important challenges to the development of distributed simulation techniques, specially in the case of highly interactive simulation.

Whether or not it is possible to go from a Smalltalk operational simulation of the timeLock algorithm as outlined in this paper to an actual implementation on a multiprocessor configuration is still an open question. However several recent positive results in actor and object-oriented language implementations, e.g. (Lalonde, Thomas and Pugh 1986). allow us to be confident that the presented description is quite close to a possible distributed implementation scheme.

Bibliography:

Bézivin, J. & Imbert, H. (1982) Adapting a simulation language to a distributed environment. Proceedings of the 3rd International Conference on Distributed Computing Systems, Miami, Florida, (october 1982), pp.596-605.

Bézivin, J. (1987) Some experiments in object-oriented simulation. OOPSLA'87, Orlando, Florida,(october 1987)

Birtwistle, G.M. (1979) DEMOS A System for Discrete Event Modelling On Simula. The MacMillan Press, LTD, 214 p.

Goldberg, A. & Robson,D. (1983) Smalltalk-80: The language and its implementation. Addison Wesley, 714 p.

Hoare, C.A.R. (1974) Monitors: an operating system structuring concept. CACM, V.17, N.10, (october 1974), pp.549-557

Jefferson, D.R. & Sowizral,H. (1985) Fast Concurrent Simulation Using the Time Warp Mechanism. Proc. Conf. on Distributed Simulation 1985, (january 1985), San Diego, pp.63-69.

Lalonde,W.R. & Thomas,D.A. & Pugh,J.R. (1986) Actors in a Smalltalk multiprocesor: a case for limited parallelism. SCS-TR-91, School of Computer Science, Carleton University, Canada, (may 1986), 6p.

Misra, J. (1986) Distributed Discrete-Event Simulation. Computing Surveys, Vol. 18, N.1, (march 1986), pp.39-65