Proceedings of the 1990 Winter Simulation Conference
Osman Balci, Randall P. Sadowski, Richard E. Nance (eds.)

A UNIFIED DISTRIBUTED SIMULATION SYSTEM †

Jeff McAffer

Defence Research Establishment Ottawa, Ottawa, Ontario, K1A 0Z4
and
School of Computer Science, Carleton University, Ottawa, Ontario, K1S 5B6

ABSTRACT

We propose a method for generalizing existing distributed simulation algorithms such as, Time Warp and Chandy-Misra to create a Unified Distributed Simulation algorithm. By explicitly defining *risk* and *aggressiveness* parameters for each model, models with different behaviours can be mixed within one simulation. We illustrate how this results in a more powerful environment for creating complex simulations. Current distributed simulation techniques are presented and contrasted. We relate computational reflection and speculation to the Unified Distributed Simulation algorithm and detail the concurrent object-oriented programming environment created to support Rival, its implementation.

1. INTRODUCTION

Simulation is an increasingly important computational tool. Computer simulation has been characterized by Nobel laureate Ken Wilson as the third branch of science, which complements theory and experimentation [Bell 1989]. This view was also voiced by several speakers at the recent *Grand Challenges to Computational Science* conference [Levin 1989]. As the size and complexity of experimental systems increases, scientists are unable to provide adequate formal specifications for these systems because component interactions appear random and complex, and system topologies highly dynamic. Simulation mechanisms must be more flexible and take better advantage of available computing resources.

Distributed computing is becoming recognized as a good architecture for attacking large computational problems [Bell 1989; Bézivin 1987]. It offers high performance, flexibility and scaleability at a reasonable cost. Unfortunately, distributed computing concepts cannot be immediately applied to the simulation of complex systems. Current simulation technology is either inadequate for modelling complex systems or cannot utilize the power of distributed computers. The main goal of this work is to create a tool for building logical, flexible simulations which can be run efficiently on a distributed computer system.

1.1 Models

A model is a description of a real-world, or proposed system. A model can be expressed in different ways depending on its intended use. For example, a model of a bank from the point of view of customer service would describe the behaviour of the various queues and servers in the bank. A transaction bank model, on the other hand, models the bank as a series of transactions, each of which must be processed atomically as in the real transaction system. Combining the two bank models into one simulation is difficult because their components are behaviourally dissimilar. The interaction in the service view is predominantly sequential and services are not interrelated. The transaction model, on the other hand, is a highly asynchronous, distributed system which requires a detailed description of the processing system's operations and concurrency control. Currently, the simulation system best suited to the service model is not well suited to the transaction model and vice versa. Therefore, combining the models requires that one or the other be re-implemented in an inappropriate environment and subsequently reverified and revalidated.

The cost of verification and validation increases rapidly as the quantity and complexity of models and their interactions rises. The capability to combine behaviourally different models will lower these costs by allowing more model reuse and reducing the requirement for reimplementation. Costs can also be reduced by increasing the separation between the simulation specific code and the model specific code.

1.2 Time

Directly or indirectly, time plays a significant role in almost all systems. It is hard to conceive of an entirely time independent system. The role of time may be anything from that of a simple history ordering mechanism (e.g., something happened at time T), to a constraining sequencer (e.g., event A must occur before event B), to a proactive agent in the system (e.g., at time T some event occurs). Therefore, the timing mechanisms used in a simulation are of the utmost importance, affecting not only a model's design and implementation, but what models can be simulated and the performance of the simulation.

Uniprocessor time maintenance (e.g., time- and event-driven simulation) is straightforward because the entire system is centralized and only one object can be simulated at any given time. In these, systems time is implemented as a global variable. This is unacceptable for distributed simulation.

1.3 Distributed Simulation

The introduction of multiprocessing and communication between concurrent components complicates time maintenance. Throughout the simulation, communicating objects must be explicitly synchronized in simulation time. Uniprocessor simulation methods (e.g., centralized event queueing and single stepping clocks) rely on shared global memory and do not exploit parallelism inherent in the models. We show that for a powerful and flexible distributed simulation system, a sound model of concurrency is more important than the use of actual multicomputer hardware. Further, such a model of concurrency is inherently speculative and is best implemented in a computationally reflective environment.

The goal of this work is to continue work done previously [McAffer 1989a, b] and create a distributed simulation algorithm which unifies current techniques (e.g., Chandy-Misra and Time Warp) and results in a scaleable, flexible and responsive simulation system. Although we believe that such a Unified Distributed Simulation [McAffer 1990] system will be faster for many simulations, our main goal is to develop a simulation system in which a model's behaviour can be changed dynamically either through reflection or external intervention and models can use whatever simulation technique and time coordination mechanism best suits their structure and behaviour.

† The algorithms and intellectual property contained in this document may be subject to the applicable patent laws of Canada, the United States, and other countries in the name of Her Majesty the Queen in the Right of Canada as represented by the Minister of National Defence.

1.4 Layout

The remainder of this paper is organized into four sections. Section 2 contains a brief review of current distributed simulation concepts and technology. Section 3 presents the Unified Distributed Simulation algorithm. Section 4 discusses reflective computation and speculation relative to UDS and Rival, a realization of UDS. Section 4 also includes a simple example which highlights the features of UDS. The final section summarizes our findings, presents some preliminary results, and provides some directions for future work.

2. DISTRIBUTED SIMULATION SYSTEMS

A distributed simulation system must explicitly coordinate the advance of time in order to maintain temporal consistency between the components. The system must define how time is advanced when, according to the models, it should be advanced. Thus, time maintenance can be divided into two distinct tasks; the movement of time and the coordination of time movement. In this paper we are concerned with the coordination of time advances between concurrent simulation components. We believe that each component in the system should specify its own method of synchronizing with the rest of the simulation.

In the following sections we present two distributed simulation algorithms which typify the state-of-the-art in distributed simulation. We do not attempt to summarize the entire technology but rather illustrate the diversity of the approaches. Both of these techniques use the same notion of local and global virtual time. The local virtual time, LVT, of an object is defined as the time up to which that object has simulated. So, for example, if an object has completed all of the operations assigned to it up to time 238 then its LVT is 238. Depending on the time coordination technique used, a particular LVT may be monotonic or non-monotonic. The global virtual time, GVT, of a simulation is analogous to the LVT of a component. It is the point in simulation time up to which all components have successfully simulated. The GVT at a particular point in the simulation is equal to the minimum of all the LVTs in the system. If some component, M, has LVT equal to the GVT then it is said that M *defines* the GVT.

2.1 Chandy-Misra

Chandy-Misra (CM) simulation was the first distributed simulation algorithm and is the result of work by several different groups [Peacock, Wong and Manning 1979; Chandy and Misra 1981]. For convenience we use the name Chandy-Misra even though some of the properties expressed did not originate from their work.

Chandy-Misra is a form of *pessimistic* simulation. The mechanism holds back processing because it assumes that components will communicate out of sequence. CM can be viewed as a token passing mechanism in which an object with a token is free to communicate with any other object in the simulation while those without tokens may do only local processing. An object gets a token when it defines the GVT (i.e., when its LVT equals the GVT). Since more than one object can define the GVT, more than one object can have a token. When there are no more objects with tokens, the GVT is updated to be the minimum LVT of all objects in the simulation and the tokens are redistributed. In this way, the GVT is guaranteed to equal the least LVT of all interacting objects in the simulation.

In CM, synchronous message passing is used to coordinate concurrent components. Messages cannot be received until the receiver's LVT = GVT, that is, the message's sender remains blocked until the receiver defines the GVT. This restriction guarantees that messages are received in the correct order. However, forcing objects to wait for LVT = GVT is excessive. Since most objects interact with a limited set of neighbors, a receiver need only wait to process a message until its neighbor's times are greater than the message time. Unfortunately, this requires all components to maintain lists of neighbors. The use of such lists restricts the ability of the simulation system to model complex dynamic systems. If the list is static then it will contain all possible neighbors and will force objects to do excessive synchronization. Maintenance of dynamic neighbour lists is complicated and requires additional synchronization whenever the graph topology is changed.

The basic Chandy-Misra algorithm does not prevent simulation induced deadlock. That is, deadlock which is due solely to the simulation process. Deadlock detection, avoidance and recovery is a difficult problem and has received much attention in both the simulation and general distributed/concurrent computing literature. These methods include *null message* passing [Chandy and Misra 1979], demand-driven null message passing [Misra 1986] and a detection-recovery scheme [Chandy and Misra 1981; Misra 1983; Kumar 1986].

CM works best in tightly coupled simulations (i.e., simulations in which objects are highly synchronous) which are highly connected. In these systems, communicating objects are usually synchronized and thus seldom have to wait for their messages to be delivered. The performance of CM simulations is bounded above by the critical path of concurrency present in the simulated model. It is not possible for events which are synchronous in the real system to be processed asynchronously in the CM simulation of that system.

2.2 Time Warp

Time Warp (TW) [Jefferson 1985; Jefferson and Sowizral 1985; Jefferson, *et al* 1987], sometimes referred to as *optimistic* simulation, relies on the ability of an object to *rollback* its present state to that of some previous time. In contrast to Chandy-Misra, objects using Time Warp simulate and communicate freely with other components until they receive a message. When an object receives a message it must synchronize with the message's sender. If the new message is from the receiver's future (i.e., if the message timestamp is greater than the receiver's LVT) then the receiver increases its LVT to the message timestamp value and processes the message. If the message is from the component's past then the receiver must undo, *rollback*, any processing it did or messages it sent between its current LVT and the new message's time.

One's initial reaction to the concept of Time Warp might be that the simulation will continually take nine steps forward only to take eight steps back. Jefferson and Sowizral [Jefferson and Sowizral 1985] postulated that the time spent projecting an object's future is not really wasted since, in a scheme like Chandy-Misra, the interaction would be blocked for synchronization and the simulation would not progress. In fact, only the objects involved in that particular interaction will be blocked and the processors simulating those objects will still be available to run the other objects in the simulation. In addition, since all message passing is asynchronous, Time Warp is entirely free of simulation induced deadlock. This simplifies algorithms and eliminates the overhead previously required to detect or avoid deadlock.

The rollback process can be improved through the use of *lazy cancellation* [Gafni 1985; Gafni, Berry and Jefferson 1987]. Under this model, *antimessages* are sent only when it is clear that the corresponding messages should never have been sent. Berry [Berry 1986] proved that the performance of Time Warp using lazy cancellation is not bounded by the critical path of synchronization. This result is demonstrated in [Berry and Lomow 1987].

Time Warp is good for loosely coupled, highly asynchronous systems but is inefficient when models have mixed time scales or diverse interaction behaviours. The interested reader is referred to [Lavenberg, Muntz and Samadi 1983; Gilmer 1988; Lomow *et al* 88] for performance analyses of Time Warp and heuristic rules for optimizing snapshot frequency and rollback costs.

2.3 The Optimism Spectrum

Simulations written using optimistic techniques are quite different from those written using pessimistic methods. There are several systems for example, Moving Time Window (MTW) [Sokol, Briscoe and Wieland 1988] and Bounded Lag (BL) [Lubachevsky 1988, 1989]) which attempt to address the middle ground between optimism and pessimism. Unfortunately, they have not been fully successful. We agree with Reynolds [Reynolds 1988] when he says that there is a range or spectrum, "of different possibilities and that it is worth considering points in this range which are between the two extremes". Furthermore, we believe that transitions between points in this spectrum should be seamless and that components functioning at different points should be compatible. That is, optimistic and pessimistic simulation should be *unified.*

Reynolds [Reynolds 1988] presents nine design variables which can be used to characterize current distributed simulation systems. These variables are; Partitioning, Adaptability, Aggressiveness, Accuracy, Risk, Knowledge embedding, Knowledge dissemination, Knowledge acquisition and Synchrony. In this paper we will consider only *risk* and *aggressiveness*, and add one of our own, *compatibility.* Aggressiveness is the property of processing messages based on conditional knowledge. That is, relaxing the requirement that messages be processed in a strict monotonic order with respect to message times. Risk is passing messages which have been processed based on aggressive or inaccurate processing assumptions in a simulation component. Compatibility is the ability of components in a simulation to have different bindings for their design variables. For example, can one model be aggressive while there are others in the simulation which are not?

In light of this we see that Chandy-Misra simulations are non-aggressive while Time Warp simulations are maximally aggressive. Similarly, Chandy-Misra and Time Warp treat risk as a discrete variable with one of two values, 0 or ∞, respectively. While MTW and BL are partially aggressive and admit a certain amount of risk depending on their aggressiveness, the variable values are not dynamic and cannot be defined for each object. Furthermore, none of these systems is compatible.

Current technology is incapable of simulating dynamic, complex systems which contain a mixture of synchronous and asynchronous components. Consequently, simulationalists are forced to work around the simulation tools to implement their models. Much of this effort can be avoided by unifying the simulation techniques. That is, by creating a system which is adaptable, which allows continuous and infinite levels of aggressiveness and risk, is 100% accurate and allows individual models to determine how they will interact. In addition, the simulation system must include a sound model of concurrency and all models must be compatible. Our goal is to develop such a system.

3. UNIFIED DISTRIBUTED SIMULATION

The Unified Distributed Simulation algorithm (UDS) [McAffer 1990] is loosely based on the Time Warp algorithm. UDS can consistently model both optimistic and pessimistic components as well as models with limited bidirectionality. UDS parameterizes each component's level of optimism by its *aggressiveness* and *risk.*

The aggressiveness, A_i, $(0 \le A_i \le \infty)$ of a component, M_i, is the number of time units M_i is capable of looking ahead. The range $[GVT, GVT + A_i]$ defines a *receive window* within which M_i is capable of receiving and processing messages. Therefore, M_i is responsible for rolling back any non-committed actions taken during this period (i.e., simulation for times after GVT).

The risk, R_i, $(0 \le R_i \le \infty)$ of a component, M_i, is the number of time units into the global future M_i is capable of sending messages. The range $[GVT, GVT + R_i]$ defines a *send window* within which M_i is capable of sending messages although it is up to the destination to determine if the message will be received.

The aggressiveness and risk parameters allow UDS to use *variably asynchronous message passing.* That is, the message $\langle T_{send}, T_{arrive}, M_j, M_i \rangle$, where M_j and M_i are the sender and receiver respectively, can only be sent if T_{send} is in M_j's send window and received if T_{arrive} is in M_i's receive window. Otherwise, M_j will block on either the send or receive of the message. The reader will note that UDS is similar to the Moving Time Window concept [Sokol, Briscoe and Wieland 1988] in which there is only one receive window for all components and no send windows.

Figures 1 through 3 show message queues for models with different sized time windows. In these figures, the arrow indicates the next message to be read from the queue. Messages have three states; processed (white), received (shaded) and sender blocked (black). Notice that as the aggressiveness increases so does the number of messages processed in the projected future.

In UDS, a model's processing capabilities are regulated by the size and position of its send and receive windows. Each model, M_i, is provided with a continuously updated estimated GVT, or EGVT (M_i), which defines the origin of its windows. The inequality EGVT $(M_i) \le$ GVT always holds. In figure 2 we see that the message at time 49 is blocked. This is due to the inaccuracy of EGVT (Y). If EGVT (Y) = GVT = 20, then the message would be received. The UDS algorithm allows A_i and R_i to vary dynamically and has the property that as the A_i and R_i decrease, the required accuracy of EGVT (M_i) increases. If EGVT (M_i) is sufficiently inaccurate then it is possible that M_i's message queue will be erroneously empty and M_i will stop processing. Simulation induced deadlock can occur if this were to happen to all of the models simultaneously.

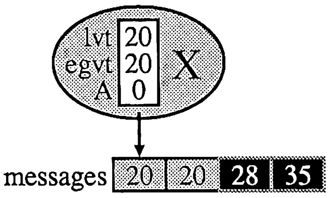

Figure 1. Snapshot of Message Queue Where A = 0

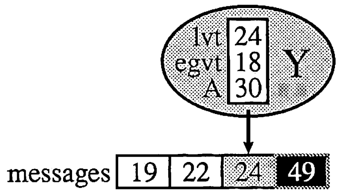

Figure 2. Snapshot of Message Queue Where A = 30

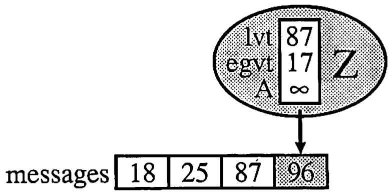

Figure 3. Snapshot of Message Queue Where A = ∞

If at all times the origin of all time windows is equal to the GVT (e.g., EGVT (M_i) = GVT, for all i) and there exists no message, $\langle T_{send}, T_{arrive}, M_{source}, M_{dest} \rangle$, such that $M_{source} \ne M_{dest}$ and $T_{arrive} - T_{send} > A_{dest}$ where A_{dest} is the aggressiveness of the destination, then fatal simulation induced deadlock cannot occur [McAffer 1990].

Because the GVT is dynamic and distributed, the algorithm for calculating the EGVT cannot guarantee that EGVT (M_i) = GVT at all times. In practice it is enough that the EGVT algorithm be capable of calculating the exact GVT when deadlock occurs. In general, EGVT (M_i) can be inaccurate by as much as $(GVT + A_i) - T_{arrive}$ and messages at time T_{arrive} will still be received and processed by M_i.

4. IMPLEMENTATION

We have implemented UDS in ENVY™/Actra, [OTI 1990; Thomas, LaLonde and Pugh 1986] a multiprocessor Smalltalk/V™

[Digitalk 1986] system running on MC680x0 processors on a VMEBus system. Actra runs on top of the Harmony Real-Time Operating System (RTOS) [Gentleman 1985] which provides basic facilities for interprocessor communication and multiprocessing. The current system runs with four processors however, the UDS algorithm allows for any number of processors to be used. A uniprocessor implementation of UDS has also been done using Smalltalk/V on a PC.

4.1 Smalltalk

We have used the Smalltalk/V object-oriented programming system to implement the UDS algorithm. In Smalltalk everything is an *object*. Every object is an *instance* of some *class* which is also an object. Classes describe the structure and behaviour of their instances while the instances contain the data. An object's behaviour is the set of *methods* or procedures which the object can execute. Methods are invoked by sending a message containing the method's *selector*, the method name, and any required arguments, to an object. If the behaviour of the object receiving the message defines that method then the message is processed, otherwise an error occurs. Since messages are sent to objects rather than sending objects to procedures, a variable's class is irrelevant as long as it can understand the specified message. Another important part of Smalltalk is the inheritance or subclassing mechanism. Subclasses *inherit* and refine the structure and behaviour of their superclasses. The *structure* of a subclass is the accumulation of the structures defined by itself and its superclasses while the *behaviour* of a subclass is given by the union of the methods defined by itself and its superclasses.

4.2 Speculative Computation

The UDS algorithm belongs to a class of computation known as *speculative computation*. Speculative computation is computation which is initiated before it is known that the result will be required [Burton 1985; Halstead 1986; Baker and Hewitt 1977; Osborne 1989]. Conversely, *mandatory computation* is computation which is known to be required. Using speculation, idle processor time found in non-speculative systems is used to compute possible future results. Unified Distributed Simulation is a speculative algorithm in which the amount of speculation possible is proportional to the aggressiveness and risk of the various objects. Similarly, Chandy-Misra is non-speculative and Time Warp is fully speculative.

Speculative systems fall into one of three categories; *algorithmic, code-based* and *system-based.* The UDS system performs algorithmic speculation. Code- and system-based speculation can be used in UDS but they are not essential to the algorithm. In effect, each object in a UDS simulation is a separate algorithm which projects its own future based on the common data set provided by the other components. Therefore, each algorithm speculates and contributes to the final result of the simulation. The main problem is that the data set is dynamic and all components have global side-effects. UDS can limit the number of side-effects by restricting the spread of speculation (i.e., reducing the risk) or limiting the initiation of speculation (i.e., reducing the aggressiveness). However, a reduction of either parameter will also reduce the potential for parallelism.

The literature contains little on the use of general algorithmic speculation. We surmise that this is because of the difficulty of undoing the widespread effects of interacting speculative computations. It is beyond the scope of this paper to explore the possible contributions of UDS to general speculative computation. However, we envisage a distributed speculative scheduler which uses the risk and aggressiveness parameters to control task scheduling and parallelism at an algorithmic level. The scheduling duties could be distributed to the tasks by increasing their reflective capabilities and allowing them to adjust their own synchronous behaviour as required. We must look at the effects of risk and aggressiveness settings with respect to processor use before such a scheduler could be deemed feasible or useful.

4.3 Computational Reflection

Computational reflection is the process of doing computation about computation. Typical computations model some real or abstract entity, say a database, which we call the application. The application in a reflective system is the computation itself. That is, reflection is the capability to look at the computation from the level of the machine which is running it. This capability can be used to dynamically modify the behaviour of the system. There are several systems which have reflective capabilities including; 3Lisp [des Rivieres and Smith 1984] and ABCL/R [Watanabe and Yonezawa 1988].

ABCL/R is a concurrent object-oriented language which supports the notion of meta-objects. For each object, A, there is an one-to-one mapping to a meta-object, $\uparrow$A. That is, $\uparrow$A describes A, just as A describes some entity in the problem domain. A is called the *denotation* of $\uparrow$A. Meta-objects are similar to Smalltalk classes, in that they define both the structural and computational aspects of their denotation. However, they differ in that A and $\uparrow$A execute in parallel and there is a unique meta-object for each object. As a result, $\uparrow$A can change the behaviour of A while A is executing. Because both A and $\uparrow$A are objects, there also exists an $\uparrow\uparrow$A, the meta-object for $\uparrow$A. This infinite chain of meta-objects is similar to the infinite tower of interpreters found in 3Lisp. *Level shifting* between the interpreters is done automatically when a meta-object receives a message. Thus, reflective procedures for an object are implemented in its meta-object. The utility of computational reflection is demonstrated in [Watanabe and Yonezawa 1988], in which a simple implementation of Time Warp in ABCL/R is described. The entire mechanism is implemented by redefining, in the meta-object, the way an object receives messages.

4.4 Rival

Rival, the Smalltalk implementation of UDS is motivated by ideas from computational reflection and actor theory [Agha 1986; Agha and Hewitt 1987]. Our definition of the term actor deviates from its original use. For our use, an actor is a group of cooperating objects which functions independently of, and asynchronously to the other actors in the system. In Actra, an actor is a Smalltalk object with additional mechanisms for multiprocessing and communications. The message passing protocol used is based on the send, receive, reply primitive set found in Harmony. The behavioural similarities of actors to objects and the simplicity of their protocol results in a powerful programming environment which can be used to create an ABCL/R style model of concurrency.

Figure 4 shows a concurrent meta-object structure implemented using actors. Every component in a Rival simulation is an actor. Figure 4 shows two objects, their associated meta-objects and their communications patterns. Objects communicate by sending synchronous actor messages to each other's meta-objects. The meta-object dictates how objects send and receive messages. All interactions between components of the simulation can be described in this way. Notice that a single message from B to A requires three actor messages, one from B to $\uparrow$B, one from $\uparrow$B to $\uparrow$A and one between A and $\uparrow$A. This overhead would be unacceptable in a production system but for our purposes it is more important that we have control over the interprocess communications.

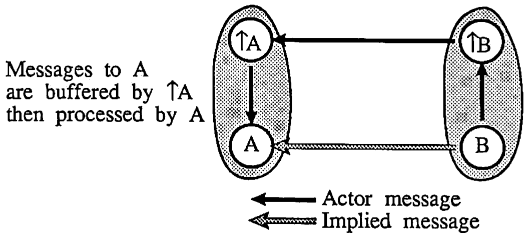

Figure 4. Objects and Meta-Objects

Unlike fully reflective systems, Rival's reflective capabilities are limited to inter-object communication and thus a level shifting interpreter is not required. Components are highly reusable because each can define its own concurrent behaviour and external protocol. Models can be exchanged between simulations even if their internal definitions are vastly different.

The class hierarchy of the complete Rival system is shown in figure 5. Classes in *italics* are new and are required for actors while those in **boldface** implement the Unified Distributed Simulation system. All other classes are supplied by Smalltalk. The discussion below details only those classes specifically related to the UDS algorithm (i.e., those in boldface).

- Object
  - Collection
    - ...
      - SortedCollection
        - **IndexedSortedCollection**
  - Process
    - *Task*
      - *Actor*
        - **GVTEstimator**
        - **MetaObject**
        - **SimulationObject**
  - **SimulationMessage**

Figure 5. The Rival Class Hierarchy

4.4.1 SimulationMessage

The instance variables in SimulationMessage are: `source`, `destination`, `kind`, `body`, `sendTime`, `arrivalTime`, `startTime` and `endTime`. To interact in a simulation, objects send *request* messages (i.e., SimulationMessages with `kind = #request`) to each other. The `startTime` of a request is the time, local to the destination, at which processing of the request began. Similarly, the `endTime` is when the destination finished processing the request. Accordingly, if the request has not been processed, the start and end times are undefined. These time fields are used by components to detect if a rollback is required and if so, the time to which they should rollback.

4.4.2 IndexedSortedCollection

Instances of IndexedSortedCollection (ISC) are used to implement the time-bounded infinite message queues in which request messages are stored in ascending `arrivalTime`-order. An ISC's index points to the next request to be read. The protocol of the ISC `add:` method is very important here. `add:` should return true if the new element is inserted before (in time) the ISC's next element index. Also, the algorithm used to insert elements guarantees that when a duplicate element (e.g., a message with an `arrivalTime` the same as a message already in the queue) is added, it will be added after those elements to which it is equal. Figure 6 shows an example where a message, E, having timestamp 25, is added to a queue. Its sorted position is immediately following message B which also has timestamp 25. This positioning is guaranteed by the algorithm. Notice that the queue's index or *next element* has been moved to point to E, making it the next message in the queue. In this example, `add:` will return true.

4.4.3 GVTEstimator

The GVTEstimator is an actor which polls the components in the simulation for estimates of the current GVT and sets each component's EGVT to the minimum of these values. This technique is suitable because the UDS algorithm requires only that the EGVT be less than or equal to the real GVT. The accuracy of the GVTEstimator's result is inversely proportional to the amount of simulation which takes place during one round of polling. Additionally, the simulation has the property that as the GVT estimate decreases in accuracy, more components will suspend processing and more processor time will be available to the GVTEstimator. These two properties balance each other and keep the simulation running at a reasonable rate while avoiding deadlock. If for some reason the simulation does deadlock, the GVTEstimator is, by default, given exclusive use of the processors and will generate an exact value for the GVT and resolve the deadlock. It should also be noted that the more optimistic a component is, the larger its time window and the less accurate the EGVT must be to keep the simulation running.

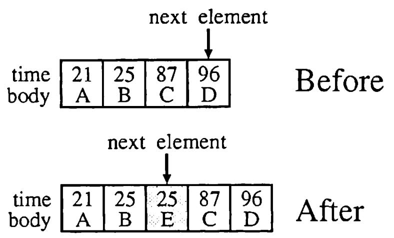

Figure 6. Insertion of a Duplicate Element

4.4.4 SimulationObject

SimulationObjects have the following instance variables: `lvt`, and `stateQ`. Each object in the simulation is modelled by an instance of a subclass of SimulationObject (SO). A SimulationObject describes the domain specific behaviour of a particular component. Each SimulationObject has an associated meta-object which is an instance of MetaObject. When the SimulationObject is ready to process a request, it invokes the `receiveRequest:` method defined by its MetaObject and processes the returned request. If there are no requests, the object is blocked, otherwise `receiveRequest:` supplies the next request.

SOs support the Restoration phase of rollback by maintaining the `stateQ`. Whenever its EGVT is updated, a SO garbage collects the `stateQ` and ensures that the `stateQ` always contains an entry with timestamp equal to the EGVT. When a rollback is triggered, the SO restores itself from the snapshot with the latest timestamp less than or equal to the rollback time.

4.4.5 MetaObject

In Rival, each SimulationObject has an associated MetaObject (MO) through which all messages are sent and received. MOs are also responsible for garbage collection, maintenance of the EGVT, coordination of rollbacks and maintaining the receive and send windows. The following are the instance variables of MetaObject which are relevant to our discussion; `requestQ`, `outQ`, `lvt`, `egvt`, `risk` and `aggressiveness`. The protocol for MetaObject which is of interest here is; `gvtEstimate:`, `request:`, `requestAccepted`, `sendRequest:`, `receiveRequest:` and `undoRequest:`.

When a MetaObject, say M, receives a request, R, its `request:` method first checks if the `arrivalTime` is within its receive window as defined in section 3. If so, M accepts R by sending the `requestAccepted` message to request's sender, otherwise the sender is blocked until M's receive window advances to include R. Then M adds R to the `requestQ`, an instance of IndexedSortedCollection. If when R is added, it is inserted before the next request in the queue (i.e., if `add:` returns true) then R is out of sequence and should have been received and processed earlier in simulation time. M must rollback to the latest possible time for which R can be processed consistently. Note that this *rollback time* is the `endTime` of the previous request after the new request is added to the `requestQ`, not necessarily the `arrivalTime` of R.

In the Cancellation phase of a rollback, MOs undo messages by sending an `undoRequest:` message for each request in the `outQ` which was sent after the rollback time. This may cause other components to rollback. The cascade of rollbacks which may result, is guaranteed to terminate if the rollback of one component cannot cause a rollback of some other component to an earlier time. That is, the rollback of some object, A, to 30 cannot cause some other object, B, to rollback to a time earlier than 30. This guarantee is implicit in the UDS algorithm. If lazy cancellation is being used then the requests to be undone are simply marked as cancelled and are undone, if required, at a later time.

The MetaObject method, `receiveRequest:`, performs a blocking receive waiting for a message to enter the receive window. Under normal conditions the MO returns the next available request, but if the MO has detected a rollback condition, it returns a rollback request containing the rollback time. While the SO is restoring its state, the MO carries out the cancellation phase. The object then Coasts Forward by rewinding the `requestQ` to the appropriate time and reprocessing the requests.

The MetaObject's `sendRequest:` protocol is used when SOs want to send requests to other components. The request is first logged in the `outQ` and then, if the request's send time (i.e., the current lvt) fits into the send window, it is sent immediately. Otherwise, the request is marked as pending and is forwarded to the receiver when the send window advances to include the request's send time.

A MetaObject's receive and send windows are kept up to date by the GVTEstimator. The GVTEstimator supplies a new value of the EGVT and requests the MO's estimate of the new EGVT. A MO calculates its estimate based on the elements in its `requestQ` and the state of its associated SO. The MO then uses the new EGVT to move the receive window and accept more requests, unblocking their senders. It also moves the send window and sends any pending messages. Note that in general there will be at most one pending send because the SO must block until the message is accepted by the destination.

Since a SimulationObject and its MetaObject are concurrent, many of these operations can be carried out in parallel. Also note that even if the SO part of a component is blocked, the corresponding MetaObject is still available to send and receive messages and contribute to the estimate of the GVT.

4.5 An Example: The Traffic System

In this example we present a simple system whose component interactions are diverse and non-deterministic. It is presented to illustrate the deficiencies of the current technology and show how UDS can be used to eliminate the problems.

4.5.1 The System

Consider a traffic system which contains both highways and city streets. The system model has the following basic properties. There are thousands of simple cars which require little processor power to simulate. The roads cross at intersections which can contain at most one car and allow cars to pass through in FIFO order. As such, intersections synchronize cars. When two cars attempt to enter an intersection at the same time, one is forced to wait. There are many intersections in the cities and few on the highways. Cars enter an intersection through one side and leave through any of the other sides (i.e., no U-turns). Traffic may backup from one intersection into another and thus intersections may interact. Crosswalks are a special kind of intersection which can contain both cars and pedestrians, although not at the same time. Any number of pedestrians may enter a crosswalk asynchronously and occupy it for varying amounts of time. Cars cannot enter crosswalks occupied by pedestrians and pedestrians have entrance priority over cars.

The traffic system has many different requirements for component interaction. Cars and pedestrians are asynchronous while intersections and road segments are synchronous. As the density of intersections increases, so does the amount of synchronous behaviour. Within the city there are *pockets* of synchrony (e.g., intersections containing cars) and each pocket interacts asynchronously with its neighbor.

Using Chandy-Misra simulation, the bulk of the concurrency in the system will be lost. The intersections dominate the CM simulation because they define the synchronous behaviour. CM is efficient for modelling a single intersection but will require each car to be synchronized with all intersections in a full system model. This will result in a highly connected system graph and high overhead if null message passing is used. Cars in the busiest intersections will hold back all cars, even those on the highways. In addition, CM is not capable of exploiting the concurrency between distinct heavy traffic routes within the city and between cars in the city and those on the highways.

Time Warp is better able to take advantage of the available system-wide concurrency but will perform poorly for intersections. As the number of intersections increases, cars become more and more synchronized and rollbacks more expensive. If, for example, a *late* pedestrian steps into a crosswalk, the crosswalk and all cars passing through it must be rolled back. In a city, these cars will have gone on to interact with other intersections and thus more cars and the rollback's cascade will be widespread. The cost of performing rollbacks can easily dominate the relatively low cost of simulating a car. A global multiprocessor load balancing scheme would improve performance by ensuring that only the components with the least LVTs are run by each processor. Unfortunately, such a scheme does not exist and would be difficult to implement efficiently on top of Time Warp because it would require global knowledge and/or reflective capabilities.

The main difficulty is that the components behave both synchronously and asynchronously depending on where they are in the traffic system. Lowering a car's risk and aggressiveness in the city will the number and extent of rollbacks. Increased aggressiveness and risk on the highway will take advantage of the available parallelism. Many of these problems can be overcome in this specific case but not in general. The incorporation of reflective capabilities allows models to adapt to changing requirements and resources.

4.5.2 The Results

The traffic system example is simplistic but its implementation in Rival does illustrate the potential of variable synchronism. Our example system contains Intersections and Cars. Figure 7 shows the closed road network in which the cars travel. In this figure, the nodes are intersections and the edges are roads. The numbers annotating the edges indicate the amount of time required to traverse that edge.

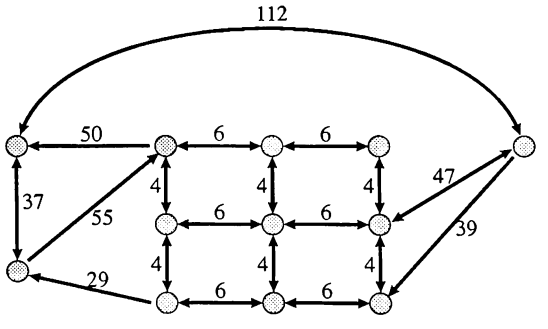

Figure 7. Example Road Network

In a particular simulation run, some number of cars are injected into the road network and allowed to roam randomly. When a car comes to an intersection, it sends the `carEntering:from:` message to the intersection. When the intersection is free, the car will be sent the `proceedAlong:` message with a random edge as the argument. The car will proceed along the edge, taking the appropriate amount of time and then begin the cycle again.

An abridged version of the source code for this example is shown in figure 8. The reader will notice that the code is very simple and straightforward and that there is no simulation specific code required. The risk and aggressiveness of the models is maintained by the simulation system. This code is shown to give the reader an idea of how Rival simulations are structured.

```
Object subclass: #Edge
   instanceVariableNames: 'start end length'
   classVariableNames: ''
   poolDictionaries: ''


SimulationObject subclass: #Car
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''

proceedAlong: anEdge

   lvt := lvt + anEdge length.
   anEdge end
      carEntering: self
      from: anEdge start


SimulationObject subclass: #Intersection
   instanceVariableNames: 'neighbors'
   classVariableNames: ''
   poolDictionaries: ''

carEntering: aCar from: anIntersection

   lvt := lvt + 1.
   aCar proceedAlong:
      (self
         randomNeighborExcept: anIntersection)
```

Figure 8. Source Code for Traffic System Example

This example has been used to perform preliminary performance testing of the UDS algorithm. Tests were carried out on a two processor system in which one processor was 30% faster than the other. In each test the simulation was run until a constant simulation time while the number of cars and the component/processor topology was varied from run to run. The risk and aggressiveness was also varied from run to run but for a specific run all components had the same risk and the same aggressiveness. In general, the test results show that performance increased when risk and aggressiveness values were between 0 and ∞. For instance, with risk and aggressiveness = 20 the simulation ran 2.5 times faster than with R and A = 0 and 2 times faster than with R and A = ∞, independent of the number of cars. These initial results indicate that varying the risk and aggressiveness can have a substantial effect on the performance of the simulation. These results are quite informal and a good deal of work must be done before any conclusive statements can be made. To carry out a proper performance study would require improved tools for monitoring and analysis. Even with these tools, such a study would be labourious because of the number of variable parameters (e.g., task/processor distribution, risk and aggressiveness).

5. CONCLUSIONS

We have presented Unified Distributed Simulation, a system which unifies optimistic and pessimistic distributed simulation techniques. It has been shown that UDS extends the functionality of the current distributed simulation techniques. UDS allows components to describe how and when they are willing to receive and process messages and as a result, gives the user more power and flexibility. In particular, UDS allows the interaction of models with explicitly different concurrent behaviours. Unified Distributed Simulation is different from existing systems since it specifies the model of concurrency as an integral part of the time coordination mechanism. This provides support for the dynamic, varyingly asynchronous, concurrent components which are required when modelling complex systems.

We have also presented Rival, a prototype system which incorporates the UDS algorithm. Rival is implemented in Smalltalk and uses ideas from computational reflection and actor theory to create a sound and flexible model of concurrency. Rival has been quite useful as an experimental tool for testing the ideas of UDS but serious use of Rival will require the addition of tools for building, running and monitoring simulations.

Since Rival integrates the optimistic and pessimistic methodologies, it is a good environment for performing efficiency comparisons between the two techniques. The mechanisms used are based largely on those of Time Warp but allow fully pessimistic operation. A good implementation of UDS should perform no worse than Time Warp or Chandy-Misra simulations. In fact, although extensive performance comparisons have not been conducted, the experiments presented in section 4 indicate that the UDS algorithm may be more efficient for some systems. Additionally, we feel that Rival should be more efficient than other distributed simulation mechanisms simply because the communication patterns more closely follow those found in the real system. This eliminates unnecessary synchronization overhead. As with Time Warp, the use of lazy cancellation allows simulations to exceed the lower bound of concurrency, but unlike Time Warp, UDS also permits this to occur in synchronous simulations.

The development and implementation of the UDS algorithm has brought out some interesting ideas relating to distributed computing in general. For instance, the use of coarse-grained speculation in distributed applications and its mixture with code- and system-based speculation. The use of finer-grained speculation promises to reduce the cost of future projection by delaying computations until they are required or there is idle processor time. This work has not addressed several areas of general distributed computing systems, for example, exception handling and application deadlock. Solutions to these problems would allow the techniques presented here to be generalized and used to create a speculative distributed scheduler in which all tasks cooperate to make the system more efficient. We believe that this is only possible through the use of sound computational frameworks such as computational reflection.

ACKNOWLEDGEMENTS

The author would like to thank Brian Barry (DREO) and Dave Thomas (Carleton University) for their invaluable assistance and advice throughout this project.

This work was carried out with the support of the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

Agha, G. (1986), *ACTORS: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA.

Agha, G. and Hewitt, C. (1987), "Concurrent Programming Using Actors", *Object-Oriented Concurrent Programming*, Yonezawa and Tokoro Eds. MIT Press, Cambridge, MA, 36-53.

Baker, H. and Hewitt, C. (1977), "The Incremental Garbage Collection of Processes", MIT Artificial Intelligence Laboratory Memo 454, Cambridge, MA, December.

Bell, G. (1989), "The Future of High Performance Computers in Science and Engineering", *Communications of the ACM* 32(9), 1090-1101, September.

Berry, O. (1986), "Performance Evaluation of the Time Warp Distributed Simulation Mechanism", Ph.D. Dissertation, University of Southern California, Los Angeles, CA.

Berry, O. and Lomow, G. (1987), "The Potential Speedup in the Optimistic Time Warp Mechanism for Distributed Simulation", In *Proceedings of the Second International Conference on Computers and Applications*, Beijing, China, 694-698, June.

Bézivin, J. (1987), "Some Experiments in Object-Oriented Simulation", In *Proceedings of OOPSLA '87*, Orlando, FL, 394-405, October.

Burton, F.W. (1985), "Speculative Computation, Parallelism, and Functional Programming", *IEEE Transactions on Computers*, C-34(12), December.

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, 440-452.

Chandy, K.M. and Misra, J. (1981), "Asynchronous Distributed Simulation via a Sequence of Parallel Computations", *Communications of the ACM* 24(4), 198-205.

des Rivieres, J. and Smith, B.C. (1984), "The Implementation of Procedurally Reflective Languages", Report CSLI-84-9, Center for the Study of Language and Information, Stanford University, CA.

Digitalk (1986), Smalltalk/V is a trademark of Digitalk Inc.

Gafni, A. (1985), "Space Management and Cancellation Mechanisms for Time Warp", TR-85-341, University of Southern California, Los Angeles, CA, October.

Gafni, A., Berry, O., and Jefferson, D. (1987), "Optimized Virtual Time Synchronization", *Applied Mathematics and Performance Models of Computer Systems*, Rome, Italy, 229-244, May.

Gentleman, M. (1985), "Using the Harmony Operating System", ERB-966, NRCC No. 24685, National Research Council of Canada, Ottawa, Ontario.

Gilmer, J.B. (1988), "An Assessment of Time Warp Parallel Discrete Event Simulation Algorithm Performance", In *Proceedings of Distributed Simulation 1988*, San Diego, CA, 45-49, February.

Halstead, R.H. (1986), "Parallel Symbolic Computing", *Computer* 19(8), 35-43, August.

Jefferson, D.R. and Sowizral, H. (1985), "Fast Concurrent Simulation Using the Time Warp Mechanism", In *Proceedings of Distributed Simulation 1985*, San Diego, 63-69, January.

Jefferson, D.R. (1985), "Virtual time", *ACM Transactions on Programming Languages and Systems* 7(3), July.

Jefferson, D.R. *et al* (1987), "Distributed Simulation and the Time Warp Operating System", *Operating Systems Review* 21(5), 77-93.

Kumar, D. (1986), Ph.D. dissertation, Computer Science Department, University of Texas at Austin, Austin, Texas.

Levin, E. (1989), "Grand Challenges to Computational Science", *Communications of the ACM* 32(12).

Lomow, G., Cleary, J., Unger, B., and West, D. (1988), "A Performance Study of Time Warp", *Distributed Simulation 1988*, San Diego, CA, 50-55, February.

Lubachevsky, B.D. (1988), "Bounded Lag Distributed Discrete Event Simulation", In *Proceedings of Distributed Simulation 1988*, San Diego, CA, 183-191, February.

Lubachevsky, B.D. (1989), "Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks", *Communications of the ACM* 32(1), 111-131, January.

McAffer, J. (1989a), "A Simulation System Based on the Actor Paradigm", Technical Note 89-4, Defence Research Establishment Ottawa, Ottawa, Ontario, February.

McAffer, J. (1989b), "Actor-based Simulation", *Proceedings of the Summer Computer Simulation Conference 1989*, Austin, TX, 910-915, July.

McAffer, J. (1990), "Unified Distributed Simulation", M.C.S. Thesis, School of Computer Science, Carleton University, Ottawa, Ontario, March.

Misra, J. (1983), "Detecting Termination of Distributed Computations Using Markers", In *Proceedings of the 2nd ACM Principles of Distributed Computing*, Montreal, 290-293.

Misra, J. (1986), "Distributed Discrete-Event Simulation", *Computing Surveys* 18(1), 39-65, March.

Osborne, R.B., (1989), "Speculative Computation in Multilisp", Ph.D. Dissertation, Laboratory for Computer Science, MIT, MIT/LCS/TR-464, December.

OTI (1990), ENVY is a trademark of Object Technologies International.

Peacock, J.K., Wong, J.W., and Manning, E.G. (1979), "Distributed Simulation Using a Network of Processors", *Computer Networks 3*, North-Holland Publishing, 44-56.

Reynolds, P.F. (1988), "A Spectrum of Options for Parallel Simulation", In *Proceedings of the 1988 Winter Simulation Conference*, San Diego, CA, 325-332, December.

Sokol, L.M., Briscoe, D.P., and Wieland, A.P. (1988), "MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution", In *Proceedings of Distributed Simulation 1988*, San Diego, CA, 34-42, February.

Thomas, D., LaLonde, W., and Pugh, J. (1986), "Actra: A multitasking/multiprocessing Smalltalk", SCS-TR-92, School of Computer Science, Carleton University, Ottawa, Canada, May.

Watanabe, T. and Yonezawa, A. (1988), "Reflection in an Object-Oriented Concurrent Language", In *Proceedings of OOPSLA '88*, San Diego, CA, 306-315, September.