Unified Distributed Simulation

by

Jeff McAffer, B.C.S.

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of the
requirements for the degree of
Master of Computer Science

School of Computer Science
Carleton University
Ottawa, Ontario
March 1990

Ó1990, Jeff McAffer

This research was supported in part by the Natural Science and Engineering Research Council of Canada and the Defence Research Establishment Ottawa.

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of this thesis
Unified Distributed Simulation
submitted by Jeff McAffer, B.C.S.
in partial fulfillment of the requirements for
the degree of Master of Computer Science

Thesis Supervisor

Chairman, School of Computer Science
Carleton University

Date

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. Models from different paradigms can be mixed within one simulation resulting 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.

Keywords: Distributed computing, distributed simulation, multiprocessor simulation, object-oriented programming, concurrent programming, speculative computation, computational reflection, Smalltalk, Actors, Time Warp

Acknowledgements

I would like to thank the following people:

Dave Thomas • not only for being my supervisor and guiding me through my thesis work but for helping me throughout my years at Carleton and for being there when I needed him.

Brian Barry • for acting as a co-supervisor and for spending long afternoons discussing time. Brian provided needed support when things were bad and left both the thesis and its author better than he found them.

John Duimovich and Glen Little • for reading and criticizing this document.

Thong Nguyen and Peter Shipton • for being good friends and all around great guys.

The Defence Research Establishment Ottawa • for allowing me the time and resources to carry out this work.

Pat and George McAffer • for being terrific parents and giving me the chance to have this education and experience.

My wife Angela ' for putting up with (or is that without) me and my ranting and for not getting mad when I was late because I was talking to Brian (again).

Contents

Acceptance Sheet **i**

Abstract **ii**

Acknowledgements **iii**

Contents **iv**

Figures **v**

Chapter 1: Simulation **1**
Models 2
Time 3
Distributed simulation 5
Thesis layout 7

Chapter 2: Distributed simulation systems **8**
Chandy-Misra 10
Time Warp 14
Bounded Lag and Moving Time Window 20
The optimism spectrum 22
Goals 26

Chapter 3: Unified distributed simulation **29**
Message 29
Simulation time 30
Systems and models 32
Unified Distributed Simulation algorithm 38

Chapter 4: Implementation **48**
Smalltalk 48
Speculative computation 51
Computational reflection 55
Rival 57
The implementation of UDS 60
SimulationMessage 60
IndexedSortedCollection 61
GVTEstimator 62

SimulationObject 63
MetaObject 65
An example: The traffic system 68
Summary of implementation 71

Chapter 5: Conclusions 74

References 76

Appendix A: The traffic example A-1

Figures

2.1 Example communications graph 11

2.2 Deadlock in CM simulation 12

2.3 (a) Triggering of a rollback 15

2.3 (b) After Restoration 16

2.3 (c) After Cancellation and Coast Forward 17

2.4 The optimism/pessimism spectrum 23

2.5 Reynolds' design variables 23

3.1 Synchronous message passing 34

3.2 Asynchronous message passing 35

3.3 Snapshot of message queue where A = 0 42

3.4 Snapshot of message queue where A = 30 42

3.5 Snapshot of message queue where A = ¥ 42

4.1 Classes and instances 49

4.2 Objects and meta-objects 58

4.3 A simple engine model 59

4.4 A detailed engine model 59

4.5 The Rival class hierarchy 60

4.6 Insertion of a duplicate element 62

4.7 Normal message sequence 66

Figures

4.8 Message sequence with rollback 66

4.9 A graphical LVT display 72

A.1 Example road network A-1

A.2 Source code for traffic system example A-2

A.3 Traffic system results A-3

Chapter 1: Simulation

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 89]. This view was also voiced by several speakers at the recent *Grand Challenges to Computational Science* conference [Levin 89]. As the size and complexity of experimental systems increases, scientists find themselves unable to provide adequate formal specifications for these systems because component interactions appear random and complex, and system topologies highly dynamic.

Functional decomposition of complex systems can be ineffective because the resultant models are too removed from the real world and are hard to validate. Simulation designers need a more direct method of modelling complex systems. It is more appropriate to decompose a complex system into its logical parts, each of which is modelled individually. The modelling of a given logical component may require that it be further decomposed, but eventually the entire system will consist of a set of *primitive* components which combine to model the system.

Distributed computing is becoming recognized as a good architecture for attacking large computational problems [Bell 89] [Bézivin 87]. 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.

To build such a tool we must first look at the concept of simulation and understand its constituents; the system model and time.

Models

A model is an abstraction of a real-world, or proposed system. Models must be *verified* and *validated*, that is, we must ensure that they correctly represent the components of the system (validation) and that their implementation executes according to their description (verification).

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 can be simulated using standard high level queueing theory. 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. The simulation technique best suited to the service model is not well suited to the transaction model and vice versa.

Therefore, combining the models using current technology requires that one or the other be re-implemented and subsequently reverified and revalidated.

The cost of verification and validation increases rapidly as the quantity and complexity of models and their interactions rises. Consequently, any reduction in the amount of verification and validation required is desirable. One way to do this is to allow different models to be combined within the same simulation framework. This reduces the requirement for reimplementation and allows the simulation components to be closer to their real-world counterparts. Costs can also be reduced by increasing the separation between the code specific to the simulation mechanism and the code for the model. This allows models to be implemented using different simulation styles (e.g., time- or event-driven). Typical simulation technology does not allow models using different styles to be combined and simulationalists are forced to adapt their models to the simulation system.

Time

Time is the function which relates simulation models to the real world. Without the notion of time there would be no way to compare the behaviour of a simulated system to that of a real-world system. Time's role may be anything from that of a simple ordering mechanism, to that of a driving force behind an object's existence. The simulation mechanism must ensure that time is consistent for every model. That is, if the system model specifies that Ælower landing gear, landæ is the only correct sequence of events then the simulation mechanism must prevent Æland, lower landing gearæ from occurring.

The mechanisms for maintaining a consistent notion of time throughout a simulation are quite varied. Uniprocessor time maintenance (e.g., time- and event-driven simulation) is straightforward because the entire system is centralized and only one object can be run at any given time. In these, systems time is implemented as a global variable.

In a time-driven system, simulation is carried out in a lock-step fashion where at each step of the clock, all components are asked to perform a known set of operations. These operations update the state of the associated models and thus advance the simulation to the new current time. When all of the processing for a particular time has been completed, the simulation time is incremented by a fixed amount and the procedure is repeated. Determining an efficient fixed increment for the simulation clock is impossible when the system being modelled contains objects with different time granularities. If the time resolution is too fine then processor time will be wasted updating values which have not changed significantly. If the resolution is too coarse, it is possible for important model states to be stepped over resulting in an inaccurate simulation.

In contrast, event-driven simulation systems maintain a centralized event list in which events are sorted in ascending time order. During the simulation, a single scheduler retrieves the next event and dispatches it to the appropriate objects for processing. In general, the processing of one event will result in the posting of more events in the event list. Timestamping and sorting of events ensures that events are processed in the correct temporal order. The value of the global simulation clock is defined as the timestamp of the first event in the event list. The event-driven simulation mechanism is better and more powerful than time-driven simulation as every significant

state of a component is processed and only those states which are significant are processed. In fact, time-driven simulation is a special case of event-driven simulation in which the only events are ticks of the clock.

The introduction of multiprocessing and communication between concurrent components complicates time maintenance. Communicating objects must be explicitly synchronized in simulation time because processors have different speeds and some events take longer than others to process. This synchronization must be done throughout the simulation. Uniprocessor simulation methods (e.g., centralized event queueing and single stepping clocks) are not suitable in a distributed environment because they rely on shared global memory and do not exploit parallelism inherent in the models. More sophisticated methods such as Chandy-Misra [Chandy and Misra 81] and Time Warp [Jefferson 85] [Jefferson, *et al* 87] are required if the capabilities of distributed systems are to be utilized. Chapter 2 provides the reader with a review of different distributed simulation mechanisms and their properties.

Distributed simulation

Simulated models are becoming larger and more detailed, and correspondingly demand more computational resources and flexibility from simulation mechanisms. For example, when immunologists learn of the existence of a new antibody, they want to know what it does and how it affects its host. This can be simulated but requires a model of the host's immune system, a vast and complex network of interacting organs and cells. Such a model can only be created and run in a powerful simulation system with a sound model of concurrency and component interaction.

Similarly, the advent of relatively inexpensive, powerful processors and distributed simulation systems makes it less expensive to prototype through simulation than in hardware. This is because we are building complex systems and using expensive experimental materials. In many cases the materials required for the physical system are unavailable for much of the system's design phase. Running simulations on distributed hardware also allows for the use of heterogeneous special purpose processors (e.g., database engines and signal processors) or hardware-in-the-loop simulation. Heterogeneous processor use also allows previously developed models to be included without incurring porting or revalidation costs. This is done by adding the model's original host machine to the distributed system and implementing the required communications protocols. Since the model's implementation was not changed there is no requirement for revalidation. Similarly, man-in-the-loop simulation can be done by creating the appropriate man-machine interface.

These properties allow a distributed simulation system to be used as a testbed in the development of a complex system. As separately developed components become available they can be included in the simulation and thus the design of the system. Overall system design takes on a stepping stone flavour of producing small sub-systems and testing them in a partially simulated, partially real system. Further, domain experts can work on the components in their area of expertise separately and combine components to model the system.

Many of these concepts are quite general and do not require actual distributed processing hardware but rather a sound model of concurrency. We show that for a powerful and flexible distributed simulation system, such a model of concurrency is inherently speculative and is best implemented in a computationally reflective

environment. We will also show that current distributed simulation technology falls short in most of these areas and that this problem can be viewed as a concurrent speculative scheduling algorithm. As such, our results are applicable to distributed computation in general.

The goal of this work is to continue work done previously [McAffer 89a] [McAffer 89b] and create a distributed simulation algorithm which unifies current techniques and results in a scaleable, flexible and responsive simulation system. In such a Unified Distributed Simulation system, 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. In addition, simulations can be run on any number of processors.

Thesis layout

The remainder of this thesis is organized into four chapters. Chapter 2 contains a review of current distributed simulation concepts and technology and an analysis of the properties which can be used to differentiate one simulation technique from another. Chapter 3 presents formal definitions and specifications of the components required by the Unified Distributed Simulation algorithm. Chapter 4 discusses reflective computation and speculation relative to UDS and Rival, a realization of UDS. The reader is given a detailed description of the Rival system itself and its implementation in Smalltalk. Chapter 4 also includes a simple example which highlights the features of UDS. The final chapter summarizes our findings and provides some directions for future work.

Chapter 2: Distributed simulation systems

Directly or indirectly, time plays a significant role in many systems. 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.

Traditionally, simulation techniques have been classified by the mechanism used to signal the advance of time, for example, discrete event-, time- or process-driven. In addition, simulations have been restricted by the requirement that the same mechanism be used by all models in the simulation. Further, these systems typically have a single clock with which all models coordinate. This type of simulation is adequate for simple uniprocessor systems but falls short in a distributed environment. Frequently accessed dynamic global variables (e.g., the simulation clock or event list) cause performance bottlenecks. These shared structures must be distributed to take advantage of available parallelism.

Statistical analysis of stochastic systems and modelling of large complex systems are two example uses of distributed computers for simulation. Stochastic simulations

imply some sort of statistical analysis and as such, one run of the simulation is not enough. Rather, the results of many runs must be combined and analyzed to determine the behaviour of the system. Distributed computing can be used for this type of system by replicating the simulation several times and run them concurrently. In this case, each instance of the simulation is largely independent and there is little idle processor time.

Our work in this thesis is more concerned with modelling large complex systems for which the output is dependent on the input (i.e., data driven). More specifically, we are concerned with the mechanisms used to build and run these simulations. Distributed computing is of benefit here because it can provide large amounts of computing power at a reasonable cost.

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 thesis we are concerned with the coordination of time advances between concurrent simulation components. Current distributed simulation techniques require that all parts of the simulation use the same coordination strategy. This restriction is unwarranted and will reduce the clarity of the resulting models by forcing them to conform to the simulation tool. Each component in the system should specify its own method of synchronizing with the rest of the simulation.

In the following sections we will present four distributed simulation algorithms. Together they typify the state-of-the-art in distributed simulation. All 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.

Chandy-Misra

Chandy-Misra simulation was the first distributed simulation algorithm and is the result of work by several different groups [Peacock, Wong and Manning 79], [Chandy and Misra 81]. For convenience we use the name Chandy-Misra (CM) even though some of the properties expressed did not originate from their work.

Chandy-Misra 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. It requires that the receiver wait until its time is less than that of *all* other components in the simulation. Since most objects interact with a limited set of neighbors, this waiting time can be reduced. To do this we form a directed system graph in which the nodes are the components and the edges represent the communications links directed towards message receivers. By annotating the edges with the time of the most recent message, an object can establish a lower bound on the LVTs of all its potential senders. As this lower bound will always be greater than or equal to the actual GVT, the waiting time will be reduced.

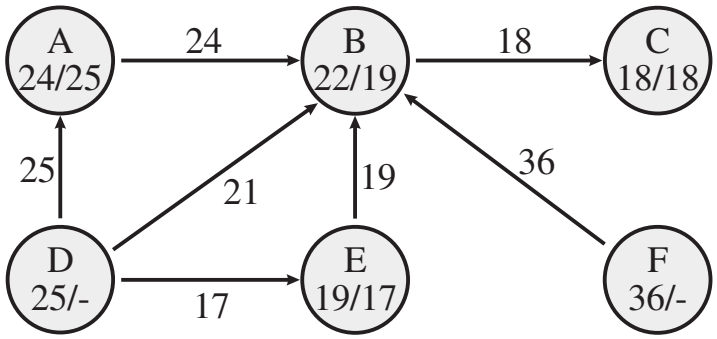

Figure 2.1 Example communications graph

Figure 2.1 contains an example system digraph. The leftmost number in a node is the component's LVT while the rightmost is an estimate of the minimum of its potential senders' LVTs. For example, B's current LVT is 22 and the minimum of its incoming channels is 19. Rather than wait for the GVT to equal its LVT, a receiver need only wait until its lower bound estimate is greater than or equal to its LVT before processing a message. The use of a digraph restricts the ability of the simulation system to model

complex dynamic systems. If the graph is a static structure then it will contain all possible edges and individual objects will be forced to synchronize with more components than are actually required. Maintenance of a dynamic communications graph is complicated and requires additional synchronization whenever the graph topology is changed.

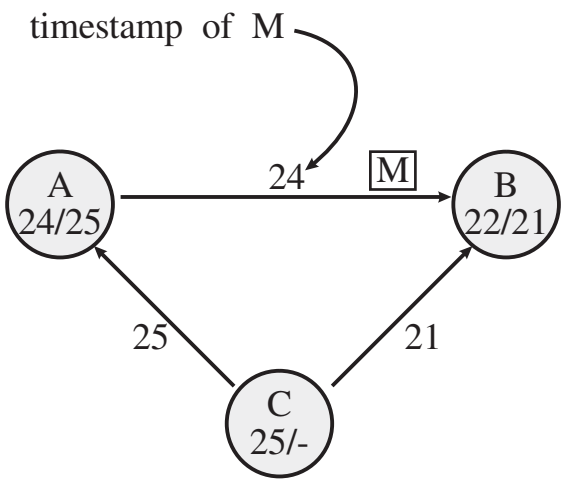

Figure 2.2 Deadlock in CM simulation

The basic Chandy-Misra algorithm does not prevent simulation induced deadlock. Simulation deadlock can occur when one object attempts to synchronize with another object which is in turn trying to synchronize with it. In figure 2.2 deadlock is caused by B's inability to receive the message, M, from A timestamped 24. For B to receive M, it would have to advance its LVT from 22 to 24. This is not possible since the channel time of CB is 21. If B were to blindly advance its time to 24 it would run the risk of C sending it a message timestamped 23. B must wait for the time on channel CB to advance. In this particular case however, C is blocked on a message send to A and will

not advance the time of channel CB. Similarly, A cannot process C's message until A's message to B is processed. The result is deadlock.

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 works fall into two general categories; guaranteed deadlock avoidance, and deadlock detection and recovery.

In figure 2.2, deadlock occurred because the edge values (e.g., CB) were not up to date. This can be prevented by introducing a message whose sole purpose is to update a link's time, a *null message*. A component receiving a null message simply updates the link time, discards the message and continues processing. Chandy and Misra [Chandy and Misra 79] [Misra 86] show that deadlock can be successfully avoided by sending a null message over all outgoing channels every time a real message is sent. If null message passing is used for the system in figure 2.2 then the time for the edge CB would be have been updated to 25 when C sent its message to A. This would have allowed B to advance its LVT to 24 and receive A's message. The deadlock would have been avoided.

Depending on the topology of the system being simulated, this null message passing may cause unacceptable overhead. Demand-driven null message passing [Misra 86] is a more flexible approach. In our example, B would ask C, the object holding it back, if the time on channel CB can be updated to be greater than or equal to 24. Since C's time is 25 it would increase CB's time to 25 and the deadlock would not occur. However, this is not always possible as C may be held back by the time of one of its incoming channels, XC. Just as B asked C, C must ask X to update the time of XC. This chain of update requests may grow until a cycle is formed and deadlock occurs. Misra points out that resolution of this deadlock is non-trivial and requires the calculation of the

minimum *next-event-time* and LVT for all components in the deadlocked cycle[1]. The deadlock is resolved by advancing the LVT of the component with the least LVT to the minimum of the next-event-times.

Chandy and Misra [Chandy and Misra 81] also propose a simulation system in the CM style which allows the simulation to deadlock, detects the deadlock and then resolves it. This system is based on the premise that in certain communications topologies deadlock avoidance is more expensive than detection and recovery. The algorithms involved in this technique are non-trivial and require topographical information relating to component interactions. Detailed discussion of the techniques or algorithms applied is beyond the scope of this thesis and the interested reader is referred to [Misra 83] and [Kumar 86].

Chandy-Misra simulation is a form of pessimistic simulation. The mechanism holds back processing because it assumes that components will communicate out of sequence. Ironically, 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.

Time Warp

Time Warp [Jefferson 85] [Jefferson and Sowizral 85] [Jefferson, *et al* 87], sometimes referred to as optimistic simulation, relies on the ability of an object to

[1]The deadlock chain may include all components in the simulation.

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 between its current LVT and the new message's time.

To facilitate rollback, each object must maintain additional state, in particular: a state queue (stateQ), a queue of all the messages it has sent (outQ) and a queue of all the messages it has received (inQ). The stateQ is a time ordered list of *snapshots* of the component. Any one of these snapshots must be sufficient to rebuild the entire state of the object[2] at its related point in time. The contents and function of the inQ and outQ are self explanatory.

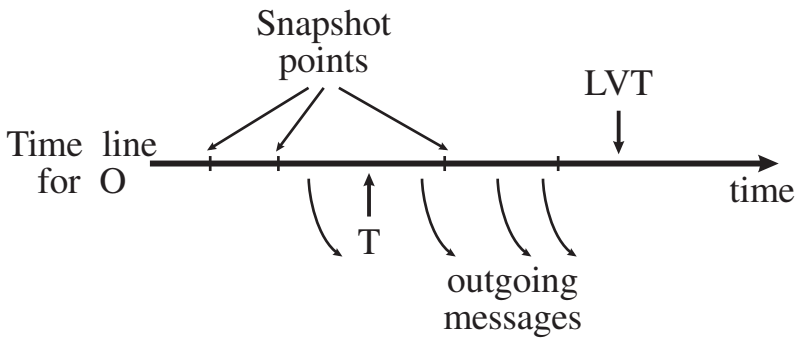

Figure 2.3 (a) Triggering of a rollback

[2]The snapshots need only contain the non-redundant information relevant to the model. Data related to the simulation mechanism need not be saved.

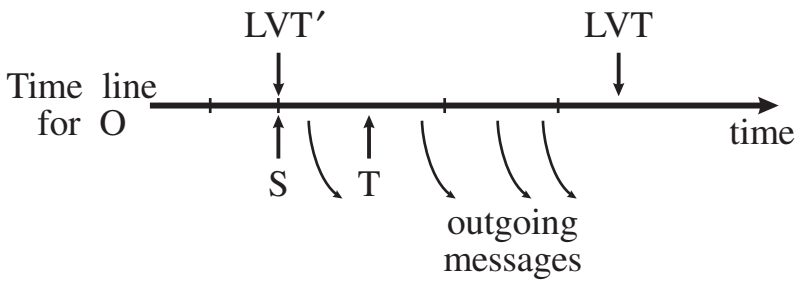

Figure 2.3 (b) After Restoration

In figure 2.3 (a) a rollback is triggered by the receipt of a message timestamped T by an object, O, whose LVT is greater than T. Rollback consists of three distinct phases, the first being Restoration. Figure 2.3 (b) shows that during this phase, O, and thus its LVT, is returned to the time, S, of the latest known state before T. Time S, or LVT′, may be less than T because a snapshot may not have been taken at time T. Any difference will be made up in the subsequent phases of the rollback algorithm: Cancellation and Coast Forward.

Cancellation undoes or cancels each message, M, sent between T[3] and LVT with an antimessage, M′. When M′ arrives at its destination, either the corresponding M has been processed by the receiver or it has not. If M has not been processed then it is simply removed from the message queue. If M has been processed then the receiver, C, must be rolled back to undo the effects of processing M. Note that the rollback of C will follow the same algorithm as the rollback of O which triggered the sending of M′. As such, C's rollback may cause other components of the simulation to rollback. In principle, every object in the simulation can be affected though in practice this is unlikely. See Chapter 4 for a discussion of cascading rollbacks.

---

[3]Note that time T, not time S is used here.

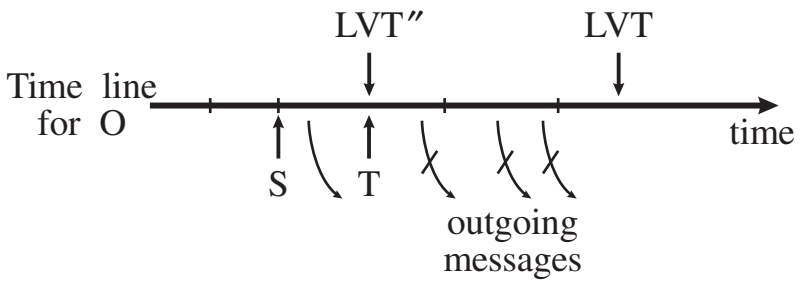

Figure 2.3 (c) After Cancellation and Coast Forward

When the Cancellation phase is complete, the simulation, exclusive of O, will be as it was at time T. Since S may be less than T, the desired rollback time, O must reprocess the messages between time S and T to regain the correct state. This is the final or Coast Forward stage. During the reprocessing of these messages it is important that O not actually send any messages as these messages have already been sent and were not cancelled in the Cancellation phase. Figure 2.3 (c) shows that after the Cancellation and Coast Forward phases, O has been returned to its original state at time LVT″ = T and is able to process the new message and maintain the correct temporal ordering of events.

The notion of GVT plays an important role in Time Warp simulation. The GVT is used to perform garbage collection of the stateQ, inQ and outQ. Since rollbacks are caused by the receipt of a message from an object's past and the GVT is defined as the minimum of all the LVTs, rollbacks to before the GVT are impossible. Therefore, there is no need to maintain the object's state and message queues for times before the GVT. As the GVT increases, old entries in these queues can be removed, freeing memory.

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. This depends on the system being simulated. Jefferson and Sowizral [Jefferson and Sowizral 85] have

observed that the amount of computational resources discarded when a projected future is rolled back is not necessarily prohibitive. The cost of rolling back a projected future depends upon the connectedness of the simulation components, the memory requirements of all remembered objects with state, the cost of the actual computations being done and the frequency of state changes in the entities. Additionally, it is 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. By viewing discarded computations and time spent rolling back as unused processor time, the two simulation schemes can be compared by measuring their processor usage.

The rollback process can be improved through the use of *lazy cancellation* [Gafni 85] [Gafni, Berry and Jefferson 87]. Under this model, the mass cancellation of messages sent between T and the original LVT is not required. Instead, an antimessage, M′, is sent only when it is clear that the corresponding message, M, should never have been sent. This can only be established once the re-evaluation of the simulation has passed the point at which M was originally sent. If the re-evaluation did not cause M to be resent then M′ is sent, cancelling M. Once a lazy cancellation occurs, the simulation must operate in this mode until the LVT is greater than any previous LVT. This process introduces a small amount of complexity to the system but saves many rollbacks.

Berry [Berry 86] 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 87] which gives examples in which Time Warp

consistently performs better than the critical path. Speedups of this nature are attainable because, under lazy cancellation, it is possible for a correct message to be sent for the wrong reason. This correct message will not be cancelled during rollback and as a result, the message receiver will have received and perhaps processed the message well before it should have been available, according to the critical path.

One of the significant benefits of Time Warp simulation is that it is entirely free of simulation induced deadlock. Since all message passing is asynchronous, there is no blocking and there can be no deadlock. This simplifies algorithms and eliminates the overhead previously required to detect or avoid deadlock.

Time Warp is inefficient for models with mixed time scales. For example, in a simulation of a banking system, the natural time scale for transaction entry is in terms of minutes or seconds while transaction processing at the central computer may be described in terms of microseconds. In this case, the rollback of a transaction can cause the loss of up to one million one microsecond simulation steps for the transaction processor. This may be many real hours of simulation run time.

In general, Time Warp is a good method for loosely coupled, highly asynchronous systems. The size and complexity of the components drives up the cost of snapshots and rollbacks and thus the cost of the simulation. If the cost of component computations is expensive relative to the cost of the rollbacks then Time Warp is efficient. All of these factors significantly affect the number of objects which can simultaneously exist in the simulation as well as the simulation's speed. The interested reader is referred to [Lavenberg, Muntz and Samadi 83] [Gilmer 88] [Lomow *et al* 88] for performance analyses of Time Warp and heuristic rules for optimizing snapshot frequency and rollback costs.

Bounded Lag and Moving Time Window

Bounded Lag (BL) [Lubachevsky 88] [Lubachevsky 89] and Moving Time Window (MTW) [Sokol, Briscoe and Wieland 88] were developed independently, but both work on the principle of limited optimism. They allow asynchronous execution within the bounds of some parameter, P. For Bounded Lag simulation this parameter is the bound and for MTW it is the window width. In this section we will describe the BL and MTW algorithms and point out their differences.

Both algorithms are based on the premise that a simulation can be broken up into segments or regions of concurrency. The parameter P defines the size of these segments and the GVT defines their positions in time. As the GVT increases, the simulation moves from region to region asynchronously processing the events in the current region. In BL, the GVT is updated whenever the events for the current region are exhausted or when it is estimated that one object in the region will affect the history of another. In MTW, the GVT is updated by a task which polls the components and determines a value for the GVT.

For MTW the size of the region or time window is very important. If the window is large enough to contain more than one event for the same object then time anomalies are possible. A time anomaly is when a component receives an event from its past. In Time Warp simulation an anomaly triggers a rollback to restore the correct ordering of events. MTW does not allow rollbacks so if an anomaly occurs, the simulation results may be inaccurate. MTW attempts to control anomalies by reducing the size of the window and explicitly synchronizing when particular types of messages are sent. [Sokol, Briscoe and Wieland 88] details the constraints on controlling anomalies and some

possible methods for automatically adjusting the size of the time windows to reduce or eliminate the problem.

The implementation of the time window concept in the MTW simulation environment is complicated and restrictive. Anomalies are unacceptable in large complex simulations as they prohibit the verification and validation of models. Attempts to limit or eliminate anomalies will introduce significant overhead and complexity to the algorithm and reduce overall flexibility. These problems are not easily resolved within the current MTW system.

The Bounded Lag algorithm is more centralized and synchronous than MTW and does not suffer from time anomalies. The algorithm contains a number of synchronization steps at which each component must stop and wait for the others before continuing to process events. These synchronization steps are used to delimit regions of asynchronous behaviour within the algorithm. For each object, BL calculates the next time, T, at which the object will be affected by the other objects. It then simulates all events between the present (GVT) and min (GVT + T, GVT + P). The GVT is then recalculated. Each phase of the algorithm is carried out asynchronously with synchronization occurring between phases. According to this algorithm, as the frequency of object interaction increases so does the amount of blocking synchronization and the number of times the GVT must be recalculated. In highly connected systems this overhead will become excessive.

Bounded Lag requires knowledge about the system it is simulating in order to function efficiently. In particular, non-zero estimates of the minimum message propagation delays are needed. These estimates are used to predict the component interactions for a portion of the simulation. The propagation delays are precomputed,

static values, one for each path between a pair of components in the simulation. The calculation of the propagation delays requires a directed graph of the system's communication channels. As with CM, this requirement all but precludes the modelling of systems in which components are dynamically created and destroyed and communications links are added and deleted.

The optimism spectrum

In the preceding sections we have discussed four existing distributed simulation techniques. Most of the research to date has been on the creation, improvement and optimization of these algorithms. The literature contains little on the parameters or factors which could be used to differentiate one coordination scheme from another. In fact, the only discussion found by the author is in a paper by Reynolds [Reynolds 88].

Current distributed simulation time coordination schemes can be divided into two basic groups, *pessimistic* and *optimistic*. These terms are used to describe how a system views communication within the simulation. For example, Chandy-Misra is pessimistic because it assumes that most communication in a simulation run will occur in the wrong order (i.e, unsynchronized). Time Warp, on the other hand, is optimistic and assumes that interactions will naturally occur in the correct order. Simulations written using one technique are quite different from those written using the other and are not easily integrated. Interestingly, there are few systems like MTW and BL which address the middle ground between optimistic and pessimistic simulation.

We feel that there is a range or spectrum, to use Reynolds' words, "of different possibilities and that it is worth considering points in this range which are between the two extremes" (see figure 2.4). Furthermore, we believe that transitions between points

in this coordination spectrum should be seamless and that components functioning at different points should be compatible. That is, the two main factions of distributed simulation, optimistic and pessimistic, Time Warp and Chandy-Misra, should be *unified*.

Figure 2.4 The optimism/pessimism spectrum

DV.0: Partitioning - Determine clusters of logical processes (LPs) based on distinct sets of design variable bindings.

DV.1: Adaptability - Changing design variable bindings based on knowledge of selected aspects of the simulation state.

DV.2: Aggressiveness - 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.

DV.3: Accuracy - Requiring that events within LPs ultimately be processed in a correct (monotonic) sequence.

DV.4: Risk - Passing messages which have been processed based on aggressive or inaccurate processing assumptions in an LP.

DV.5: Knowledge embedding - Knowledge about LPs' behavioural attributes is embedded in the simulation.

DV.6: Knowledge dissemination - LPs initiate the transmission of knowledge to other LPs.

DV.7: Knowledge acquisition - LPs initiate requests for knowledge from other LPs.

DV.8: Synchrony - Degree of temporal binding among LPs.

Figure 2.5 Reynolds' design variables

Reynolds presents nine design variables which he feels can be used to characterize current distributed simulation systems. Our goals, as stated in Chapter 1, give rise to a similar set of parameters. Figure 2.5 shows the definitions of Reynolds' nine design variables (DVs) as given in [Reynolds 88]. These definitions, while generally acceptable for our work are found lacking or confining in some cases. Below we will relate the design variables to the four systems already presented and comment on their relevance and possible values in relation to the goals of our work. Note that a logical process (LP) is equivalent to a model or component.

DV.0 In practice, automatic partitioning of a complex system can only be effected by running the simulation and observing the behaviour of the models. If components are added or removed from the simulation then the partitioning must be redone (i.e., the simulation must be rerun just to formulate the partitioning). The user could supply the partitioning himself, but forcing him to do this undermines one of our fundamental goals: the user should not have to understand the simulation mechanism. None of the simulation systems presented here uses partitioning.

DV.1 As an alternative to partitioning, we propose that the underlying simulation system be able to take advantage of dynamically changing behavioural traits in the models by making models reflective. That is, any given model should be able to review its history and decide how best to behave, and prepare for the future. This capability is closely linked to adaptability. Any simulation system which allows flexible descriptions of complex dynamic systems must be adaptable. The values of the design variables must be dynamically changeable on a component basis to allow the simulator to adapt to changes in the simulated system. None of the existing systems discussed in this chapter is adaptable.

DV.2 Aggressiveness is necessary but not sufficient for a model to be optimistic. Time Warp simulations are maximally aggressive while Chandy-Misra simulations are non-aggressive. These two paradigms do not mix and each is best for particular types of models. For example, if a highly synchronous sub-system is modelled using aggressive techniques, the result will be inefficient. Systems like MTW and BL are partially aggressive in that they allow very limited amounts of future projection.

DV.3 Most simulation systems, with the exception of MTW, do not admit inaccuracies. Our view is that inaccurate simulation systems will prevent the verification and validation of models because the behaviour of the models cannot be correctly predicted and tested. As such, we do not consider inaccurate simulation techniques.

DV.4 The amount of risk and aggressiveness in a system combine to measure its optimism. Like aggressiveness, Chandy-Misra and Time Warp treat risk as a discrete variable with one of two values, 0 or ∞, respectively. BL and MTW admit a certain amount of risk depending on their aggressiveness. We will show that non-zero risk values are relevant only to systems with asynchronous message passing and that limiting risk can reduce the cost of cancelling computations (e.g., rollbacks). It is also shown that an explicit risk parameter adds to the expressiveness of the concurrency model.

DV.5 The use of embedded knowledge usually implies the existence of two things; some element of central control and some degree of static state. This can be seen in MTW and BL. In general, centralized entities in a distributed environment should be avoided as should static state which limits the simulation of highly dynamic systems.

DV.6 and DV.7 Either knowledge dissemination or knowledge acquisition is used by all simulation techniques to maintain the consistency of time and avoid, detect

and resolve deadlock. These techniques are required to help resolve contentions and allow the simulation mechanism to determine the correct course of action.

DV.8 Synchrony is only an issue when the restrictive definitions of current technology are applied to aggressiveness and risk. In a system with variable aggressiveness and risk where all components are compatible, all levels of synchrony are supported.

We would add one design variable to Reynolds' list; *compatibility*. Compatibility is the ability of LPs 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? Compatibility is essential for simulating large systems in which components are highly diverse. Current simulation technology is not compatible.

In view of these definitions we define optimistic simulations as those containing components with non-zero risk and aggressiveness, and pessimistic simulations as those in which all components have zero risk and aggressiveness.

Goals

The main goal of our work is to create a distributed simulation system which unifies as much of the variable space described by Reynolds as possible. That is, we wish to create 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 with the other models in the simulation. All models within this system must be compatible.

Current technology is, for the most part, non-adaptive and allows either full aggressiveness and risk or none at all. Many of the algorithms call for the simulation system to have explicit knowledge of model interactions and require models to proactively seek or distribute knowledge about their state during the simulation. Such systems are unacceptable for the simulation of large complex systems. As outlined in Chapter 1, we feel that complex simulations inherently contain a mixture of synchronous and asynchronous components and are typically dynamic. Consider a detailed simulation of a complete air traffic control system including aircraft movement, radar tracking and the data processing system. The simulation of aircraft movement is straightforward. Detailed modelling of the radar tracking system is more complicated since the radar system must interact with the movement model in order to track the aircraft. The most difficult and complex part is the simulation of the multiprocessor software used in the air traffic control computers. This requires a detailed model of the system tasks and the intertask communications and synchronizations as well as the actual processing of data.

A simulation of this system presents many problems. The components have vastly different time scales, they have different behavioural and interactive characteristics and they model the real world to varying degrees of detail. Current simulation technology is not able to accommodate all of these requirements and features. 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. Increasing the accessibility of the simulation tools allows domain experts with little simulation knowledge to create models of components in their domain. These components can then be combined to model complete systems. The resultant models are more accurate and easier to validate because the expert is able to implement the models himself.

Once a system model is built it should run efficiently on a distributed computer system. To facilitate this, the simulation system must include a sound model of concurrency. The model of concurrency should be explicitly stated and be at least as powerful as those found in concurrent operating environments. A strong model of concurrency also allows the simulation tool to be used as a prototyping tool providing a one-to-one mapping between the prototype and the final system. Verification and validation of the models is also made simpler because models are structurally and behaviourally similar to the real components.

Chapter 3: Unified distributed simulation

In this chapter we formalize the ideas in Chapter 2 and present the concept of Unified Distributed Simulation (UDS), a new time coordination technique for distributed simulation. This formalization will allow us to generalize the optimistic and pessimistic techniques and through this generalization, combine them into one mechanism in which all models are compatible.

The following are formal definitions of the elements involved in distributed simulation. We use the terms; *model*, *process*, *component* and *object* interchangeably to denote a discernable piece of a simulated system (e.g., an engine is a part of a car).

Message

In a simulation, messages are bounded by events. The sending of a message is an event as is the arrival of a message at its destination. Events may also cause messages to be sent and received, thus they may cause more events. A message, for the purposes of simulation, is a vector $\langle T_{send}, T_{arrive}, M_{source}, M_{dest}, \texttt{kind}, \texttt{body} \rangle$. The send time, T_{send}, is the time, relative to the source's clock, at which the message was sent. The arrival time, T_{arrive}, represents the time, relative to the destination's clock, after which the message can be processed by the destination. The inequality $T_{send} \leq T_{arrive}$ always holds since an object cannot send a message into its own past. M_{source} and M_{dest} are the

sending and receiving models, respectively. The message `kind` is a label for the message and the `body` is the message data. These two fields are not relevant to the simulation algorithm and in the discussion below, will typically not be specified. Rather, messages will be denoted by a 4-tuple, $\langle T_{send}, T_{arrive}, M_{source}, M_{dest} \rangle$, in which the field ordering is important. For example, $\langle 30, 50, M_1, M_2 \rangle$ denotes a message sent from M_1 at time 30 arriving at M_2 at time 50. If the value of a particular field is unspecified, it will contain the field label (e.g., M_{source}).

Simulation time

The notion of simulation time is much like real time. An event which happens before another event, occurs at an earlier time. That is, its time of occurrence is less than that of the second event. Therefore, to define the set of comparison operators {<, =, >}, and thus the temporal concepts of past, present and future respectively, we need only define the *happened before* relation. Simulation time itself does not necessarily have any units or represent any tangible or scaleable quantity, it is simply an ordering maintained by a clock. This clock can be logical or physical depending on the application. The clock's value is used to define when, in simulation time, events occur. Time is a scalar value which admits typical mathematical and comparative operations (e.g.; +, -, *, <, ...). Note however that the times being operated on must have the same logical units for the resultant values to be useful.

Throughout the rest of the thesis we use Lamport's notation for clocks and the total and partial orderings of events. We provide enough detail of the notation to allow its use and refer the interested reader to [Lamport 78] for more detailed rationale. Definitions 3.1 through 3.4 are due to Lamport.

Definition 3.1 Happened before relation, $\rightarrow$. The relation $\rightarrow$ satisfies the following three conditions: 1) If event *a* comes before event *b* and *a* and *b* are for the same object then $a \rightarrow b$. 2) If *a* is a message send from one component and *b* is the receipt of the same message by a different component then $a \rightarrow b$. 3) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. Note that *a* and *b* are *concurrent* if neither ($a \rightarrow b$) nor ($b \rightarrow a$).

Unfortunately, in a distributed simulation, $\rightarrow$ is sufficient only to define a partial ordering of events in a system. It is not possible to relate the events in one part of the system to those in another as $\rightarrow$ is a causal ordering not a temporal one. To allow for a total ordering then, we introduce the concept of time.

Definition 3.2 Logical clock. The logical clock, C_i, for a model, M_i, takes on a monotonic sequence of values such that an event *a* in M_i occurs at *time* $C_i(a)$. Note that this is equivalent to M_i's LVT when *a* occurred.

Definition 3.3 Clock condition. For any events *a* and *b*, if $a \rightarrow b$ then $C_i(a) < C_j(b)$ for i, j = 1, 2, ..., n where n is the number of clocks in the system.

From definition 3.1, the clock condition is satisfied if, given *a* and *b* from M_i where *a* comes before *b*, $C_i(a) < C_i(b)$, and if, given *a*, the sending of a message from M_i, and *b*, the receipt of that message by M_j, then $C_i(a) < C_j(b)$.

To define a total ordering of the events in the system we can now order them by the times at which they occur, and use an arbitrary ordering, > , of the models in the system to break ties.

Definition 3.4 Total event ordering $\Rightarrow$. Given the events *a* in M_i and *b* in M_j, $a \Rightarrow b$ if and only if either (i) $C_i(a) < C_j(b)$, or (ii) $C_i(a) = C_j(b)$ and $M_i > M_j$.

Note that the relation $\Rightarrow$ is not unique as a different ordering of the M_i would give a different relation, and that the relation $\rightarrow$ is unique and if $a \rightarrow b$ then $a \Rightarrow b$.

Systems and models

Simulation is the execution of a system of models with respect to time. A model is an approximation or description of a real-world or proposed system component and time is an ordering mechanism used to relate components to each other and to the real world. Despite the wide variety of possible component and system models, there are a number of properties which all models possess. This section presents a formal description of what a model is, how models combine to form systems and the role of time in simulations.

Process-driven simulation provides a simple mapping between the components in a distributed simulation and those in the real-world system. Chandy and Misra [Chandy and Misra 81] refer to a real-world object as a physical process (PP) and the corresponding simulated component as a logical process (LP). In keeping with terminology used previously in this chapter, we refer to a logical process as a model (M). In practice, we assume that every model M has an explicit set of processing resources (e.g., an operating system process) which is indistinguishable from M.

Definition 3.5 Model. A model, M_i, is a pair $\langle C_i, X_i \rangle$ where C_i is M_i's clock and X_i is an arbitrary state vector specific to M_i. X_i contains such things as the message queues required by M_i and any additional state required to correctly simulate the component which M_i represents to some satisfactory level of detail. Note that we use the terms *model* and *simulation model* synonymously.

In general, real-world objects can be modelled in many different ways depending on the requirements of the simulation. A system model is made up of a set of component models. Further, the simulated system is itself a model and can be included as a component in some other simulation. For the simulation to be faithful, a model, M_i, must emulate the behaviour of its real-world counterpart, PP_i. The model must correctly describe the functionality of the real component and interact with the other components in the simulation in a consistent fashion.

Definition 3.6 Consistency. A model, M_i, is said to be consistent in a system of models if an algorithm can be found which simulates the model and guarantees that M_i will not process messages from other models or allow other models to process messages from M_i in any order other than the total ordering of events in the system.

Definition 3.7 Local virtual time (LVT). The LVT of a simulation component, M_i, at some point in real time is the value of that object's clock, C_i. This is the point up to which the object has completed its simulation.

Definition 3.8 Global virtual time (GVT). The GVT of a simulation is the minimum of the LVTs of all the *active* models and the arrival times of all the unreceived messages in the system.

An active model is a model which is sending or processing a message. Inactive models do not play a role in the calculation of the GVT because they do not contribute to the progression of time. The GVT represents the point up to which the entire system has been simulated.

Definition 3.9 The properties $\uparrow$ and $\downarrow$. The property $\uparrow$ in relation to some clock C_i (i.e., $C_i\uparrow$) indicates the sequence of values produced by C_i is monotonically non-

decreasing. Similarly, the property $\downarrow$ in relation to some clock C_i (i.e., $C_i\downarrow$) indicates the sequence of values produced by C_i is monotonically non-increasing. We also characterize models by the behaviour of their clocks (e.g., $M_i\uparrow \Leftrightarrow C_i\uparrow$).

Models interact with each other by sending messages or communicating. A meaningful interaction is a set of two or more interrelated messages passed between two or more components. For example, A sends to B, B performs some processing based on A's message and after some finite time, sends a result to A. In contrast, A and B trivially interact if messages are sent in one direction only or if the contents or existence of messages is independent of any previous messages. For simplicity it is assumed that message transmission is instantaneous and models cannot send messages into their own future. Further, message processing requires a non-zero finite amount of time.

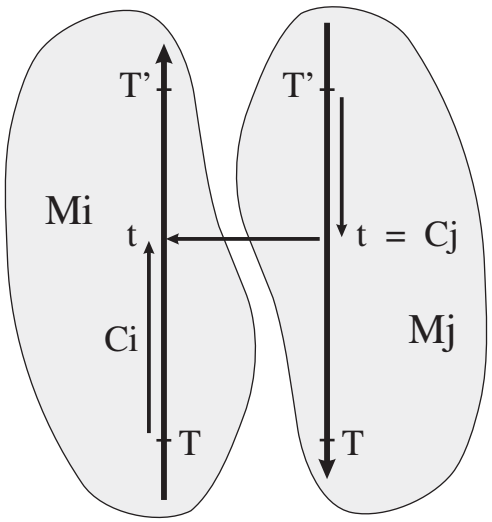

Figure 3.1 Synchronous message passing

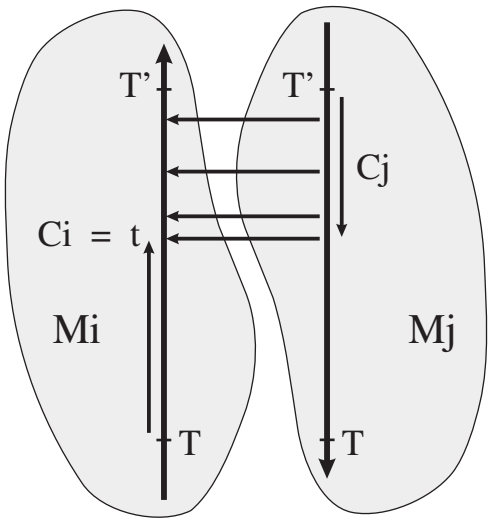

Figure 3.2 Asynchronous message passing

Lemma 3.1 Models for which time is monotonically non-decreasing cannot meaningfully interact directly with models for which time is monotonically non-increasing and vice versa.

Proof. (By contradiction). Assuming the existence of two models $M_i\uparrow$ and $M_j\downarrow$ which interact in a meaningful way, there exists an interval, [T, T′], in time over which C_i and C_j range. Two cases, synchronous and asynchronous message passing, must be considered. Assume M_j sends a synchronous message to M_i at time $t = C_j > C_i$ and the message is received and processed when $t = C_j = C_i$. See figure 3.1. M_i cannot reply to M_j because C_i is increased by processing the message. Therefore, synchronous message passing is only possible if the message receiver replies before processing the message. Further, M_j can no longer send to M_i since $C_i > C_j$ and $C_i\uparrow$ and $C_j\downarrow$. M_i and M_j can only interact through a single trivial synchronous interaction. Similarly if M_i sends to M_j. Analogously, if asynchronous message passing is used, only messages sent into the receiver's future can be received. In figure 3.2, only the messages sent by M_j *before t*

(relative to C_j) can be received by M_i. If M_i receives and processes a message from M_j then $C_i > C_j$. Even though M_i may continue to receive messages from M_j, it cannot respond to them, that is, no meaningful interaction can take place. Contradicting the assumption, neither case admits meaningful interaction. ◊

Theorem 3.1 Models for which time is monotonically non-decreasing cannot meaningfully interact indirectly with models for which time is monotonically non-increasing and vice versa.

Proof. Assume M interacts with M′. Then either M and M′ interact directly and lemma 3.1 is valid or there is a set of models, $\{M_1, M_2, ..., M_n\}$ such that $M = M_1$ and $M' = M_n$ and M_k interacts directly with M_{k+1}. By lemma 3.1, if M_1 meaningfully interacts with M_2 and $C_1\uparrow$ then $C_2\uparrow$. Similarly, if $C_1\downarrow$ then $C_2\downarrow$. Assume that this property holds for M_k and M_{k+1}. Obviously then, if $C_{k+1}\uparrow$ then $C_{k+2}\uparrow$ and vice versa. Thus, if $C_1\uparrow$ then $C_i\uparrow$ and if $C_1\downarrow$ then $C_i\downarrow$ for $1 < i \leq n$. M and M′ can communicate only if $M\uparrow$ and $M'\uparrow$ or $M\downarrow$ and $M'\downarrow$. ◊

Theorem 3.2 A system containing monotonically non-decreasing models and monotonically non-increasing models can be partitioned into two disjoint systems between which there is no meaningful interaction.

Proof. Follows trivially from theorem 3.1. ◊

Definition 3.10 Bidirectional model. A bidirectional model is a model, M_i, for which time can travel both forward and backward, that is, C_i is non-monotonic.

A bidirectional model can be consistent if an appropriate algorithm can be found to simulate the model and maintain the total ordering of events in the system. Time Warp is an example of such an algorithm.

Theorem 3.3 Given the set, $\mathbf{M}$, of all possible models in a particular system and assuming that $M_i \in \mathbf{M}$ is consistent, one can partition[1] $\mathbf{M}$ into two subsets, $\mathbf{M_I}$ and $\mathbf{M_B}$ such that time is monotonically non-decreasing for all elements of $\mathbf{M_I}$ and time is bidirectional for all elements of $\mathbf{M_B}$.

Proof. Since time can move either forwards or backwards, there are three possible temporal behaviours of a particular model; time is monotonically non-decreasing, monotonically non-increasing or both increasing and decreasing (bidirectional). Note, models for which time is constant are uninteresting. Since by theorem 3.2 non-decreasing and non-increasing models cannot meaningfully interact, $\mathbf{M}$ can be partitioned into the pairs $\langle \mathbf{M_I}, \mathbf{M_B} \rangle$ or $\langle \mathbf{M_D}, \mathbf{M_B} \rangle$ where $\mathbf{M_I}$ contains monotonically non-decreasing elements, $\mathbf{M_D}$ monotonically non-increasing elements and $\mathbf{M_B}$ bidirectional elements. Given that elements of $\mathbf{M_I}$ and $\mathbf{M_D}$ are monotonic, all C_i, where $M_i \in \mathbf{M_D}$, can be redefined to $C_i' = T_i - C_i$ where T_i is the initial time of C_i. Using this, members of $\mathbf{M_D}$ can be transformed into members of $\mathbf{M_I}$ and properties which are valid for $\langle \mathbf{M_I}, \mathbf{M_B} \rangle$ are valid for $\langle \mathbf{M_D}, \mathbf{M_B} \rangle$. Therefore, without loss of generality, $\mathbf{M}$ can be partitioned into two subsets, $\mathbf{M_I}$ and $\mathbf{M_B}$. ◊

Current technology can simulate either elements of $\mathbf{M_I}$ or elements of $\mathbf{M_B}$ but not both. Since time is strictly increasing for elements of $\mathbf{M_I}$, a simulation algorithm must be pessimistic with respect to advancing time. As such, the Chandy-Misra algorithm can be used to simulate elements of $\mathbf{M_I}$, therefore, all $M_i \in \mathbf{M_I}$ are consistent. Elements of $\mathbf{M_B}$

[1]This is not the same as the partitioning proposed by Reynolds [Reynolds 88].

allow the simulation algorithm the freedom to be optimistic when advancing time. As such, the Time Warp algorithm can be used to simulate elements of $\mathbf{M}_B$ and all $M_i \in \mathbf{M}_B$ are consistent. UDS uses the concepts presented here to generalize existing algorithms and produce a mechanism which guarantees the consistency of $M_i \in \mathbf{M}_I \cup \mathbf{M}_B$.

Unified Distributed Simulation algorithm

In Chapter 2 we discussed the merits and drawbacks of several different techniques which represent the state-of-the-art in distributed simulation. Although, theorem 3.3 shows that a system of models can be partitioned into $\langle \mathbf{M}_I, \mathbf{M}_B \rangle$ (i.e., elements of $\mathbf{M}_I$ and $\mathbf{M}_B$ can coexist), none of these algorithms can consistently simulate such a system. Chandy-Misra and Time Warp can only simulate elements from $\mathbf{M}_I$ or $\mathbf{M}_B$ respectively and neither Moving Time Window nor Bounded Lag can consistently simulate elements of $\mathbf{M}_B$. Further, the algorithms cannot be readily combined: Time Warp and Chandy-Misra have fundamental incompatibilities, MTW admits inaccuracies if the time window is too large (required by elements of $\mathbf{M}_B$) and Bounded Lag is not capable of handling bidirectional time travel. Lubachevsky [Lubachevsky 89] indicates that bidirectionality can be added to Bounded Lag, but since the algorithm allows only limited asynchronous behaviour, this bidirectionality would also be limited.

In this section we present the Unified Distributed Simulation (UDS) algorithm which can consistently simulate any system which can be partitioned into $\langle \mathbf{M}_I, \mathbf{M}_B \rangle$. Note that this is the converse of theorem 3.3. UDS has the additional property that models with limited bidirectionality can be consistently simulated. Limited bidirectionality implies that a model's clock is bidirectional but is generally increasing and is within a limited range of the simulation's global time.

The Unified Distributed Simulation algorithm is based on the Time Warp algorithm. Time Warp's rollback operation is used to support optimism and future projection. Where Time Warp requires that all models be fully optimistic, UDS parameterizes each component's level of optimism by its *aggressiveness* and *risk*.

Definition 3.11 Aggressiveness. The aggressiveness, A_i, ($0 \le A_i < \infty$) of a component, M_i, is the number of time units M_i is capable of looking ahead. M_i can simulate up to time GVT + A_i and is therefore responsible for being able to rollback any non-committed actions taken during this period (i.e., simulation for times after GVT).

Definition 3.12 Risk. The risk, R_i, ($0 \le R_i < \infty$) of a component, M_i, is the number of time units into the global future M_i is capable of sending messages. M_i can send any message with send time up to time GVT + R_i (i.e., $C_i \le$ GVT + R_i). It is up to the message receiver to determine if the message can be received.

Definition 3.13 Model (revised). An arbitrary model, M_i, originally the pair $\langle C_i, X_i \rangle$, is augmented with the values, A_i, quantifying the aggressiveness of M_i, and R_i, quantifying the risk of M_i. Thus, M_i is now defined by the tuple $\langle C_i, A_i, R_i, X_i \rangle$.

The aggressiveness and risk parameters are key to generalizing Time Warp, and unifying distributed simulation. Consider a system containing many models, one of which is pessimistic, $M_p \in \mathbb{M}_I$ and one of which is optimistic, $M_o \in \mathbb{M}_B$. Both have queues into which incoming messages are put to await processing. Assume that $C_o = C_p$ = GVT and that the first message on the queue for both M_o and M_p has arrival time T_{arrive} > GVT. M_o will immediately advance C_o to T_{arrive} and process the message regardless of the GVT. M_p must wait until the GVT $\ge T_{arrive}$ before processing the message[2]. The

[2]Remember that the inequality GVT $\le C_i$ always holds.

fundamental difference between these two behaviours is how far ahead of GVT the model can project its future. For M_p this distance is 0 while for M_o this distance is ∞.

The risk parameter describes the sending behaviour of an object. M_i cannot send a message until $T_{send} \leq GVT + R_i$. In CM simulation nothing is gained by having a non-zero risk parameter since message receivers cannot process messages in the future. Therefore, CM simulation is characterized by $A_i = 0$ and $R_i = 0$. Similarly, Time Warp can be characterized by $A_i = \infty$ and $R_i = \infty$. Explicitly variable risk allows users to describe cases where $0 < A_i, R_i < \infty$.

The addition of variable aggressiveness and risk does not invalidate any of the previous definitions or theorems, or the rollback mechanism. Consider a message, $\langle T_{send}, T_{arrive}, M_{source}, M_i \rangle$, such that $T_{arrive} < C_i$. If the rollback mechanism is faithful in restoring M_i to time T_{arrive} then processing is resumed as though no future had been projected. In the post rollback system, C_i has not generated values greater than T_{arrive} and no messages with time greater than T_{arrive} have been processed. Therefore, the next message processed by M_i will have arrival time, T'_{arrive} such that $T_{arrive} \Rightarrow T'_{arrive}$ and the total ordering is maintained.

Definition 3.14 Estimated GVT (EGVT). The EGVT is an estimate, local to a model, M_i, of the true GVT value at a given point in the run of a simulation. The inequality EGVT ≤ GVT always holds. We denote the EGVT of M_i by the function EGVT (M_i) as no assumptions regarding the derivation of the value are made.

Definition 3.15 Time window. The time window, relative to a model, M_i's incoming message queue, is the interval in time for which M_i can receive messages.

Definition 3.16 Send window. The send window, relative to a model, M_i's outgoing message queue, is the interval in time for which M_i can send messages.

Both the time window and send window of M_i are defined by an origin, typically EGVT (M_i) and an extent, typically A_i and R_i respectively. That is, the time window defines the interval [EGVT (M_i), EGVT (M_i) + A_i] in time space and the send window defines the interval [EGVT (M_i), EGVT (M_i) + R_i]. This allows UDS to use *variably asynchronous message passing*. That is, the message $\langle T_{send}, T_{arrive}, M_j, M_i \rangle$ can only be sent or received if T_{send} is in M_j's send window and T_{arrive} is in M_i's time window. Otherwise, M_j must block on either the send or receive of the message. Therefore, message passing between M_i and M_j is variably asynchronous depending on the message times, R_j, EGVT (M_j), A_i and EGVT (M_i). As such, UDS is a variant of the Moving Time Window concept [Sokol, Briscoe and Wieland 88] which defines only one time window for the entire simulation and no send windows.

The action of M_i receiving a message does not indicate that the message is processed, but rather that M_i has promised to process it consistently regardless of any other messages it receives. A model's queue is infinite in that it can hold any number of messages but time-bounded since only those messages in the time window can be accessed (received). As such, a model's processing capability is regulated by the size and position of its time and send windows.

Figures 3.3 through 3.5 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.

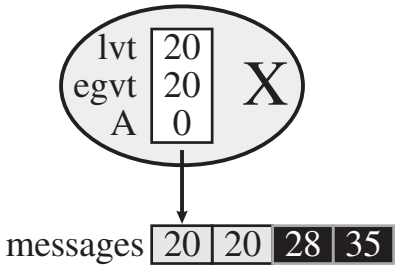

Figure 3.3 Snapshot of message queue where A = 0

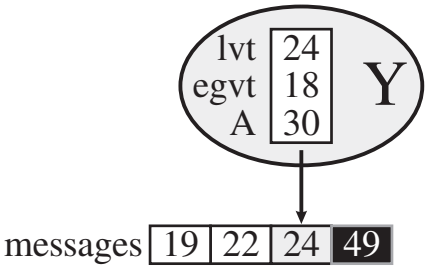

Figure 3.4 Snapshot of message queue where A = 30

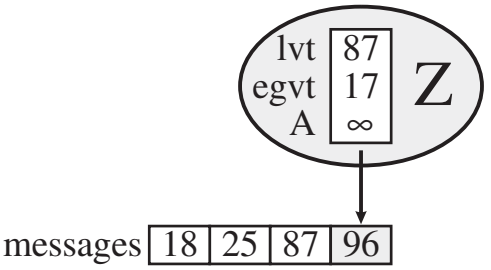

Figure 3.5 Snapshot of message queue where A = ∞

Since the position of M_i's time and send windows are dictated by EGVT (M_i), EGVT (M_i) must be reasonably accurate at all times. In figure 3.4 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 implementation of a GVT

estimation algorithm will have little impact on the UDS mechanism itself and so its description is left until Chapter 4. As the EGVT (M_i) increases, M_i's time window moves into the future allowing M_i to receive new messages and delete old ones. UDS has the property that as $A_i$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.

The send window does not have significant impact on the complexity of the UDS mechanism. As long as the EGVT is guaranteed to progress, an outgoing message is simply queued until the sender's send window includes its send time. The object is blocked but can continue to receive messages. If $R_i = 0$ then the sender will wait until it defines the GVT before sending the message. This cannot cause deadlock.

Theorem 3.4 If at all times in the simulation the origin of all time windows is equal to the GVT and there exists no message, $\langle T_{send}, T_{arrive}, M_{source}, M_{dest}\rangle$, such that $M_{source} \neq M_{dest}$ and $T_{arrive} - T_{send} > A_{dest}$ where A_{dest} is the aggressiveness of the destination, then fatal simulation induced deadlock cannot occur.

Proof. (By contradiction) Consider a deadlocked system. Then some object M must define the GVT. As such, M is sending or processing a message since objects blocked on receive cannot define the GVT. If M is processing a message, the deadlock does not exist, therefore, M must be blocked sending some message, $\langle T_{send}, T_{arrive}, M, M_{dest}\rangle$. But M cannot be blocked on the send of this message because $T_{arrive} \leq GVT + A_{dest}$ and M_{dest} will immediately accept the message. Note that an object cannot block on a message send to itself. So the system cannot be deadlocked. ◊

The significance of the restriction that $T_{arrive} - T_{send} \leq A_{dest}$ depends largely on the aggressiveness of individual simulation components and the amount of future message passing. The effects of this restriction are local to the communicating objects and will not have global consequences. Currently we restrict T_{arrive} to equal T_{send} for all messages thus the inequality $T_{arrive} - T_{send} = 0 \leq A_{dest}$ always holds. This property has no effect on any message's T_{send} or the value of R_{source}, the risk of the sender.

According to theorem 3.4, the EGVT, the real origin of all time windows, must always be equal to the GVT if deadlock is to be avoided. Because most messages have times which are well within their receiver's theoretical time window (i.e., the window with GVT as the origin), in practice it is enough that the EGVT algorithm be capable of calculating the exact GVT when required. EGVT (M_i) can be inaccurate by as much as $(GVT + A_i) - T_{arrive}$ and the messages for T_{arrive} will still be received and processed. From this behaviour we can see that as A_i decreases, the time window gets smaller and the accuracy of the EGVT must increase if M_i is to continue processing. Similarly for R_i.

We propose an algorithm which is implemented as a separate process in the system and whose accuracy is directly dependent on the percentage of processor time allocated to it. Assigning the EGVT process the same operating system priority as the simulation components insures that as components run out of messages to process, the EGVT process will be allocated more processor time and the accuracy of the EGVT will increase. This will cause more messages to be sent and received and allow more components to continue simulating. The two properties balance each other and keep the simulation running smoothly. It is possible, however, for all objects in the system to run out of messages and result in simulation induced deadlock. If this occurs then the EGVT

process is left as the only process running and it is able to calculate and distribute the exact GVT and resolve the deadlock.

We have presented the essential elements of the UDS algorithm. The existence of this algorithm proves that if a system model can be partitioned into the sets M_I and M_B then it can be simulated consistently. As noted previously, this is the converse of theorem 3.3. The algorithm will also admit several techniques or variations which can be implemented to increase efficiency or improve performance. For instance, UDS allows the use of lazy cancellation for performing rollbacks. In addition, the aggressiveness and risk parameters of each model can be varied dynamically to eliminate idle processor time and reduce the time spent blocking and rolling back.

If A_i is increased then M_i is able to look further into the future and will behave more asynchronously. Increasing the size of M_i's time window may free some models blocked on sends to M_i. There are no other side-effects of increasing A_i. Decreasing the value of A_i to A_i' decreases the size of the time window. As a result, the queue may contain a message, $\langle T_{send}, T_{arrive}, M_{source}, M_i \rangle$, such that $T_{arrive} > EGVT(M_i) + A_i'$. Conceptually, since the message is outside M_i's time window, it could not have been received and M_{source} could not have been unblocked. However, M_i did receive the message thus it is impossible for it to guarantee that the message will be processed consistently.

This situation can be avoided by tapering the reduction of A_i or, if possible, rolling back M_{source}. By requesting that M_{source} undo then redo the message send, M_i will cause M_{source} to become blocked waiting for the resent message to be received. While this solution works, it has the potential to undo considerable amounts of processing simply to vary the parameter A_i. Alternatively, tapering the reduction of A_i will not undo

any processing but will not change A_i to A_i' immediately. Assume that the message $\langle T_{send}, T_{arrive}, M_{source}, M_i \rangle$ is the latest message in M_i's queue. If $T_{arrive} \leq EGVT(M_i) + A_i'$ then all messages in the queue are within the new time window and A_i can be immediately reduced to A_i'. If $T_{arrive} > EGVT(M_i) + A_i'$ then instead of changing the value of A_i to A_i', it is set to $T_{arrive} - EGVT(M_i)$. As $EGVT(M_i)$ increases, A_i can be decreased until it equals A_i'. In this way M_i's time window will always contain all of the received messages. Similarly for variable risk.

The benefits of dynamically variable risk and aggressiveness are unclear since this capability is not available in any current system. Chandy-Misra allows no risk or aggressiveness and Time Warp allows only infinite risk and aggressiveness. It is expected that its use will increase the efficiency in simulations of systems with dynamic topologies and communication patterns by allowing models to adapt to the changing synchronization requirements. It is also anticipated that models will have to be reflective in order to fully utilize this capability since programmers will not be able to predict the optimal aggressiveness and risk settings for each model at every point in the simulation.

This chapter has formalized many of the concepts and techniques presented in Chapter 2. In particular, we have shown that Chandy-Misra and Time Warp are algorithms which enable the consistency of models for which time is unidirectional and for which time is bidirectional, respectively. Many of the design variables proposed by Reynolds [Reynolds 88] are also formalized here. We have defined an algorithm which is accurate and in which the synchrony between objects can be varied through the use of adaptable aggressiveness and risk parameters. UDS does not require that the system be partitioned into behaviourally similar sets nor does it require knowledge about the models

in the system. UDS is more expressive and adaptable than Chandy-Misra and Time Warp and as a result, gives the user more power and flexibility.

Chapter 4: Implementation

In this chapter we discuss the computational and implementational aspects of Unified Distributed Simulation. It is assumed that the reader has at least a passing knowledge of object-oriented concepts and is familiar with Smalltalk-80™ [Goldberg and Robson 83] or Smalltalk/V™ 286 [Digitalk 86].

Smalltalk

We have implemented UDS in ENVY™/Actra, [OTI 90] [Thomas, LaLonde and Pugh 86] a multiprocessor Smalltalk system running on MC680x0 processors on a VMEBus system. Actra runs on top of the Harmony Real-Time Operating System (RTOS) [Gentleman 85] which provides basic facilities for interprocessor communication and multiprocessing. A preliminary implementation of UDS was done on an uniprocessor Intel 80386-based machine using Smalltalk/V and was then ported to the Actra system. The current system runs with four processors however, the UDS algorithm allows for any number of processors to be used.

In our design goals for UDS we stated that the system should be easy to use, open and flexible. We feel that object-oriented programming in general, and Smalltalk in particular, offer these features in a single integrated package. Although we have used

Smalltlk/V, this discussion is also valid for Smalltalk-80 and so we will use the general term Smalltalk.

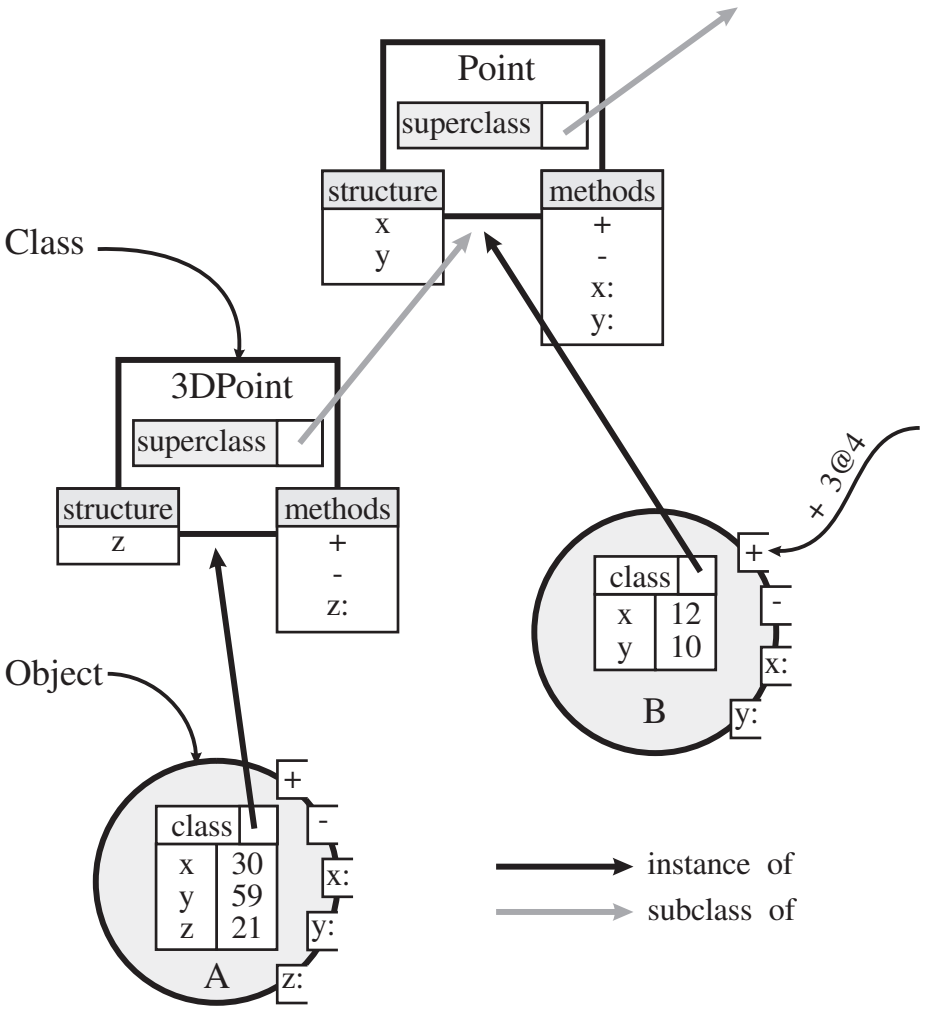

Figure 4.1 Classes and instances

In Smalltalk everything is an *object*. Every object is an *instance* of some *class* which is also an object (see figure 4.1). 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. In figure 4.1 for example, B is sent the message '+ 3@4'. B is able to respond to this message because its class, Point, defines a method for '+'. B will execute the '+' method, adding 3@4 to itself, and return the result.

This is in sharp contrast to the procedural programming paradigm where procedures are of primary importance and data are secondary. In functional systems it is up to the programmer to look at the datatypes being used (e.g., Point, Integer, Real, Fraction) and decide which of several different addition procedures to use. This impairs her ability to think conceptually and forces her to deal with implementation details. From the programmer's point of view, two numbers, say one Integer and one Real, should be added, the fact that the Integer must be converted to a Real first is irrelevant.

To implement this, Smalltalk supports late binding of variables. That is, the class of a variable is not known until runtime. Since messages are sent to objects rather than sending objects to procedures, the class of a variable is irrelevant as long as it can understand the specified message. For example, the code X + 2 will work for any X as long as the selector '+' is understood by X and its '+' method can accommodate 2 as an argument. There is no requirement for X to be a number. For example, if X were an instance of Point, X + 2 might add 2 to both the x and y coordinates of the receiver.

Another important part of Smalltalk is the inheritance mechanism. A 3-D point is type of point and therefore, 3DPoint is a *subclass* of Point. Subclasses are specializations of their superclasses: they *inherit* and refine the structure and behaviour of their superclasses. In figure 4.1 the structure of A, an instance of 3DPoint, is the accumulation

of the structures defined by 3DPoint and its superclasses. The behaviour of A is given by the union of the methods defined by 3DPoint and its superclasses. In our example, 3DPoint redefines the '+' and '-' methods, inherits the x: and y: methods and adds the z: method. The x: and y: methods can be inherited because they function the same way for Points as for 3DPoints.

Smalltalk also provides sophisticated development and debugging tools. Individual objects can be inspected and changed by the user through an interactive windowing system. Similarly, the user can debug code by interactively stepping through method statements and message sends. This facility also allows the user to modify data values while debugging and restart the execution stack at arbitrary frames. Support for user interface construction and graphical presentation is also integrated into the Smalltalk environment. Windows are objects and as such can be inspected and debugged. The inner workings of the user interface are open to the user to view and modify. All of this combines to free system designers from many of the problems of programming and allows them to concentrate on the design.

Smalltalk is an excellent environment for building powerful computational frameworks which are highly flexible. In the next two sections we will discuss some of the properties of the UDS algorithm and how general computational concepts can be applied.

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 85] [Halstead 86b] [Baker and Hewitt 77]

[Osborne 89]. Conversely, *mandatory computation* is computation which is known to be required. Using speculation, idle processor time in non-speculative systems, is used to compute possible future results. If a mandatory computation uses a speculative result then the latter is transformed into a mandatory computation. Any unused speculative computations are deleted as soon as it is known that they are not needed.

Multiprocessor systems which lack speculation can waste parallelism by unnecessarily waiting for results to become available. Speculative systems can waste processor time by projecting estimates too far into the future only to have them discarded. The distributed simulation algorithms presented in Chapter 2 exhibit these properties. Chandy-Misra performs no speculation and cannot exceed the upper-bound of concurrency. Time Warp speculates wherever possible, exploiting all available parallelism and, through the use of lazy cancellation, can exceed the upper-bound of concurrency. In a UDS or Time Warp simulation, all computation before the GVT is mandatory while all calculation after the GVT is speculative. Clearly 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.

Speculative systems fall into one of three categories; *algorithmic*, *code-based* and *system-based*[1]. Algorithmic or coarse-grained speculation is the concurrent use of several different algorithms on the same data set. Some systems use only the first result calculated and cancel all the rest. This is typically used when algorithm performance is non-deterministic or unpredictable. In these cases the various algorithms do not interact or have global side-effects. In other systems, for example, UDS and Time Warp, the results of all the algorithms are used and the different algorithms may have global side-

[1]These categories are slightly different from those put forth in [Osborne 89] but are equally discriminating.

effects. Very few systems use this type of speculation because of the difficulty of managing the side-effects when a speculation is cancelled.

Code-based or fine-grained speculation is possible wherever branching is done. It is used in many search strategies, branch-and-bound algorithms and pipeline systems. Speculation is performed by concurrently computing along all possible branch paths while calculating the branch predicate. For example, during the evaluation of an *if* statement's predicate, the system can evaluate both the true and the false cases. If the predicate's result is false then the speculation for the true case is deleted and the false case is made mandatory, and vice versa if the predicate is true. Concurrent code-based speculations with the same root (e.g., spawned by the same if statement) are independent since only one branch will be followed. That is, only one branch is mandatory, therefore, the speculative branches cannot explicitly interact.

System-based speculation is used in database managers and memory management systems. Here the system predicts the requirement for some computation based on previous operations or available information. For example, a database manager may predict that certain records will be accessed and so will fetch and cache them. This is different from code-based speculation because the speculation is loosely based on the current system state and it may happen that all of the resultant values are discarded without being used.

Notice that all three types of speculation are independent and can be combined in the same system. All three types of speculation also require the capability to cancel unwanted speculation. There are two aspects to cancelling a speculative computation; reclaiming computation and undoing side-effects. That is, killing created speculative tasks and undoing the unwanted effects of their execution. Current computation

reclamation technology is lacking [Osborne 89] although Osborne [Osborne 89] presents a strategy for undoing purely functional, non-side-effecting speculation. Osborne argues that side-effects should be used sparingly and that in most cases the complication of undoing side-effects can be avoided. Unfortunately, this is not the case in simulation where speculative components interact regularly. Note that we consider a computation to have global side-effects if it affects computations outside of its speculative branch. If a computation with global side-effects is cancelled then the side-effects must also be undone. This is analogous to rollback in Time Warp.

In a simulation, each object is a separate algorithm which may speculate on its future and interact with the other objects. Further, only a portion of a particular speculation is cancelled and all speculations contribute to the final result of the simulation. The cost of cancelling side-effects is a major drawback to Time Warp simulation. 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 cost of speculation can also be reduced through the use of *futures* [Baker and Hewitt 77] as used in Multilisp [Halstead 86a], Mul-T [Kranz, Halstead and Mohr 89] and ABCL/1 [Yonezawa, Briot and Shibayama 86]. A future is a concurrent promise to compute. The statement *future (X)* will create a task to evaluate X and will also create a *future* which will eventually hold the value of X. The future remains undefined until X has been evaluated, at which time the future is mutated and becomes the result. Even though the future may be undefined, it can be passed as an argument, assigned into variables or returned from functions. If the future is accessed before it has been defined

then the accessing task must wait for X to be evaluated before continuing. This allows the evaluation of X and the use of its result to be done in parallel.

Current implementations of the future mechanism must guarantee to evaluate X. As a result, futures cannot be cancelled and they can only be used for mandatory computations [Halstead 86a]. It is possible however to implement a future mechanism which permits explicit cancellation. Such a mechanism would limit the spread of side-effects by delaying the execution of speculative computations until their results are required. If the speculation is cancelled before then, the future is simply cancelled. Since it was not actually executed, there can be no side-effects. If the future did execute then the side-effects must be undone.

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 thesis 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 giving them some level of reflective capability 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.

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. There are two types of reflection; active and inactive. Inactive reflection is computation about computation that was or will be carried out (e.g., accessing and modifying a method dictionary). Active reflection is the ability to access the state of the current computation. That is, 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 84], ObjVlisp [Briot and Cointe 87] [Cointe 87], 3-KRS [Maes 87] and ABCL/R [Watanabe and Yonezawa 88].

The implementation of the UDS algorithm can benefit from the type of active reflection used in ABCL/R. 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, ↑A. That is, ↑A describes A, just as A describes some entity in the problem domain. A is called the *denotation* of ↑A. Meta-objects are similar to classes in that they define both the structural and computational aspects of their denotation. However, they differ in that A and ↑A execute in parallel and there is a meta-object for each object. As a result, ↑A can change the behaviour of A while A is executing. Because both A and ↑A are objects, there also exists an ↑↑A, the meta-object for ↑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 88], 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. Some additional state and behaviour is also required but is easily implemented. Since all objects are concurrent, meta-objects can be used to define a flexible model of concurrency for objects.

Current reflective systems have considerable overhead because they are typically implemented using some sort of meta-circular interpreter and so are doubly interpreted. This overhead restricts their use in practical systems. However, in general there is no reason a system cannot use limited reflection to improve its structure and flexibility.

Rival

The Smalltalk implementation of Rival is motivated by computational reflection as described in [Watanabe and Yonezawa 88] and actors [Agha 86] [Agha and Hewitt 87]. 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.

Actors are required because Smalltalk has a poor process model. Processes in Smalltalk run under the control of the Smalltalk interpreter which is itself a single operating system process. The class Process defines very little behaviour other than access and update methods and there is no explicit interprocess communication mechanism beyond semaphores.

In Actra, an actor is a Smalltalk object with additional mechanisms for multiprocessing and communications. The support for multiprocessing can be implemented either by associating a Process with each actor or by requiring that all actors be subclasses of Process. The actor communication mechanism is an extension of the

standard object behaviour implemented in conjunction with the multiprocessing support. The 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.

Actors allow programmers to explicitly define concurrent behaviour and as such, overcome the limitations of Smalltalk's process model. An actor's protocol implicitly defines its related Process' behaviour making Processes first class objects rather than data repositories. The implementation of actors in Smalltalk is simple and straightforward and gives the functionality required to realize an ABCL/R style model of concurrency.

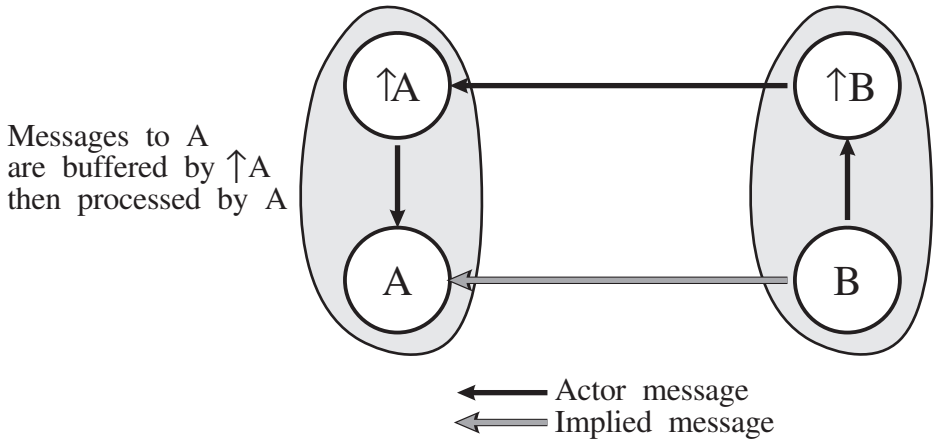

Figure 4.2 Objects and meta-objects

Figure 4.2 shows a concurrent meta-object structure implemented using actors. Every component in a Rival simulation is a concurrent entity, that is, an actor. Figure 4.2 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 ↑B, one from ↑B to ↑A and one between A and ↑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.

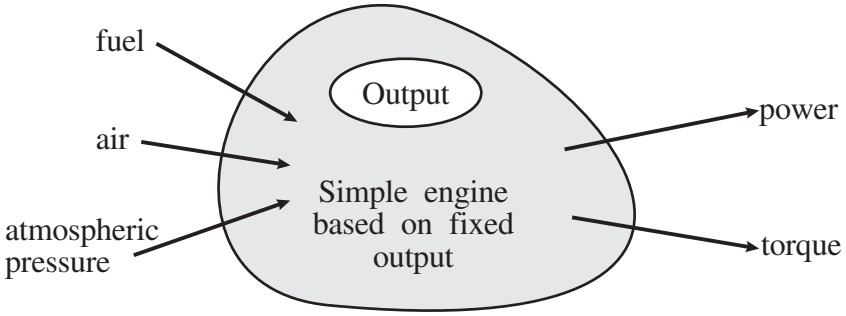

Figure 4.3 A simple engine model

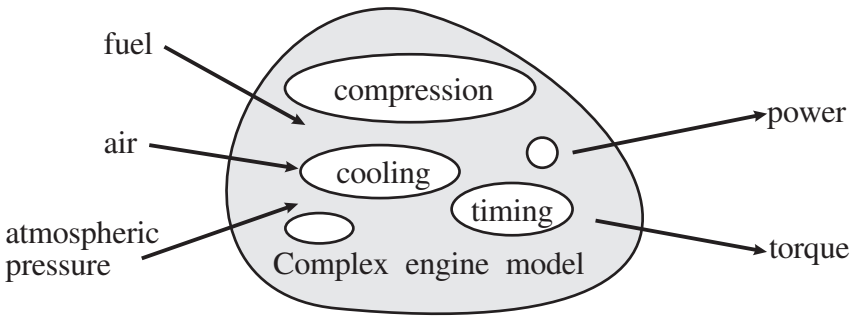

Figure 4.4 A detailed engine model

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 engine models depicted in figures 4.3 and 4.4 are interchangeable because their external behaviour is the same.

The implementation of UDS

The class hierarchy of the complete Rival system is shown in figure 4.5. 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 4.5 The Rival class hierarchy

SimulationMessage

SimulationMessage corresponds to the definition of message as presented in Chapter 3 with two additions, `startTime` and `endTime`. 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.

IndexedSortedCollection

Instances of IndexedSortedCollection (ISC) are used to implement the time-bounded infinite message queues in which MetaObjects store request messages. An ISC, as used in Rival, is a SortedCollection with a movable index and is sorted in ascending request `arrivalTime`-order. The index acts as a place marker where all elements before the index have been read and every element at or after the index have not. Implicit in this is the concept of *next* and *previous* elements where the next element is the one pointed to by the index.

The protocol of the ISC `add:` method is very important here. 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 4.6 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. The

current implementation of `add:` uses a simple, O(n), linear search to find the correct element insertion point. This can be replaced with an O(log n) binary search strategy to improve performance for large buffers.

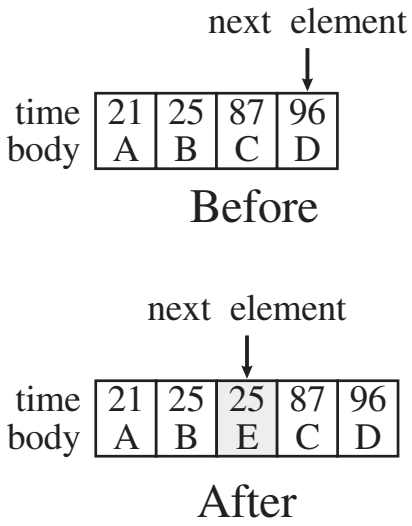

Figure 4.6 Insertion of a duplicate element

GVTEstimator

The GVTEstimator is an actor which polls the components in the simulation for estimates of the current GVT. The EGVT is the minimum of these values. Because the simulation may advance while the GVTEstimator is gathering the estimates, the EGVT may be inaccurate but since the real GVT only increases, the EGVT is guaranteed to be less than or equal to the real GVT. This is all that is required by the UDS algorithm.

When the GVTEstimator polls a component, it provides the results of the last round of polling (i.e., the current EGVT) and gets back the component's estimate of the new EGVT. Since the GVTEstimator runs concurrently with the rest of the simulation, it will get more or less time to run depending on the simulation activity. As discussed in

Chapter 3, the accuracy of the resulting estimate 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, resolving 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.

The GVTEstimator itself requires very little processor time but does perform one actor message send to each component per iteration. The majority of the time required to produce an estimate is spent waiting for components to respond to the GVTEstimator's request for an updated GVT. As the size of the simulation increases, a single GVTEstimator actor will take longer and longer to poll all of the components and generate a new EGVT. One solution is to create many cooperating GVTEstimators each of which determines the EGVT for a particular group of components (e.g., all those on a particular processor). The combined value of these estimates determines the global EGVT.

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. For example, the components in figures 4.3 and 4.4 would be

modelled as SimulationObjects. 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. If there are no requests, the object is blocked, otherwise `receiveRequest:` supplies the next request. A request can be arbitrarily complex and its processing may require the SO to send requests to other components in the simulation.

Simulation time can be advanced according to the timestamps of the incoming requests or by explicitly setting the lvt. In the first case, a SO's time is automatically advanced to the maximum of its `lvt` and the request's `arrivalTime` when request processing begins[2]. In the second, a SO can increment its `lvt` by an amount equivalent to the time required by the current operation in the real-world system.

To support rollbacks, SOs must maintain a collection of snapshots of its state, the `stateQ`. Whenever its EGVT is updated, the SO garbage collects the `stateQ` and removes elements with times less than the EGVT. These are no longer needed as components cannot be rolled back to before the EGVT. The garbage collection algorithm must ensure that the `stateQ` always contains an entry with timestamp equal to the EGVT.

When a rollback is triggered, the SO performs the equivalent of the Restoration phase of the standard rollback algorithm. This restores the SO to its state when the snapshot with the latest timestamp less than or equal to the rollback time was taken. Since the garbage collection algorithm ensures that the timestamp of the first snapshot in the `stateQ` equals the EGVT, there will always be a snapshot to restore from. In the

---

[2]The `arrivalTime` of an outgoing request must be greater than or equal to the sending SO's lvt when the request is sent. Therefore the simulation is advanced.

current implementation of Rival we require that a snapshot be taken after processing each request to ensure that for every possible rollback time[3] there is a corresponding snapshot.

MetaObject

Rival requires that objects interact in a variably asynchronous manner (see Chapter 3). To facilitate this, Rival associates a MetaObject (MO) with each SimulationObject. All messages sent and received by an SO are handled by the corresponding MetaObject methods. MOs are also responsible for garbage collection, maintenance of the EGVT, coordination of rollbacks and maintaining the time 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 time window as defined in Chapter 3. If so, then M accepts R by sending the `requestAccepted` message to request's sender, otherwise the sender is blocked until M's time 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 not necessarily the `arrivalTime` of R.

---

[3]The section on MetaObjects defines the possible rollback times.

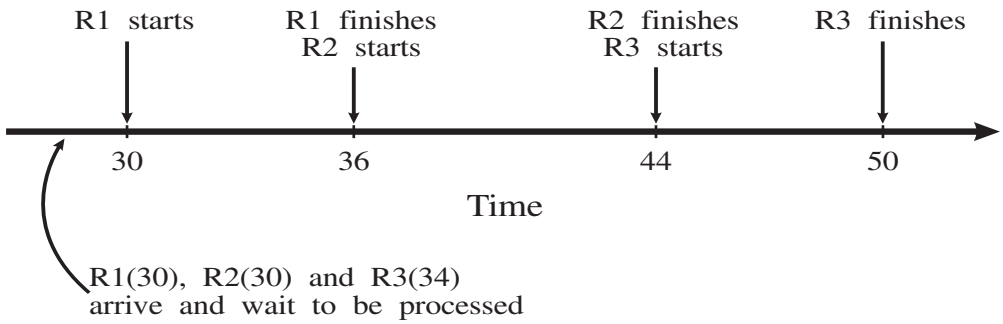

Figure 4.7 Normal message sequence

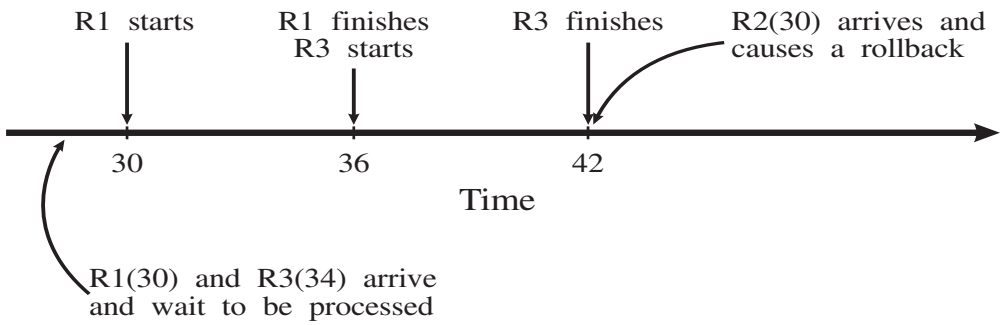

Figure 4.8 Message sequence with rollback

Consider the sequence of messages shown in figure 4.7 where the numbers in parenthesis are the `arrivalTimes` of the messages. In this case all messages were processed in the correct order. Note that the processing order of two coincident requests is non-deterministic. Now assume that R2 arrives out of sequence and causes a rollback at time 42, see figure 4.8. To what time should the object be rolled back? The `arrivalTime` of R2 is 30 but R1 was processed at 30 and the ordering of coincident requests is non-deterministic. Therefore, R2 should be processed immediately after any other messages with `arrivalTimes` less than or equal to its `arrivalTime` and the object should be rolled back to time 36, the `endTime` of R1. In general, because of the

IndexedSortedCollection element insertion algorithm's guarantees, the rollback time is the `endTime` of the previous request after the new request is added to the `requestQ`.

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. When the `undoRequest:` is received, either the cancelled request has not been processed, in which case the receiving MetaObject simply removes it from the `requestQ`, or it has been processed. In the latter case, the destination is rolled back to before the request's `startTime` and the cancelled request is removed from the `requestQ`. This rollback 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.

MetaObjects define the `receiveRequest:` protocol for SimulationObjects. `receiveRequest:` performs a blocking receive waiting for a message to enter the time 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 Coast Forward phase is performed 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 time 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 and replies the estimate to the GVTEstimator. The MO then uses the new EGVT to move the time window and accept more requests. It also moves the send window and sends any pending messages which are contained in the new window. Elements of the `requestQ` and `outQ` with times before the new EGVT are garbage collected. The newly accepted messages are made available for processing and their senders are unblocked and allowed to continue simulating. Note that in general there will be at most one pending message because the sender 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.

An example: The traffic system

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

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 act as synchronizing agents. As the density of intersections increases, so does the amount of synchronous behaviour. A car travelling into a city goes from relatively free or asynchronous behaviour to restricted or 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 to before this event. 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.

The main difficulty is that the components behave both synchronously and asynchronously depending on where they are in the traffic system. In UDS, the car's risk and aggressiveness can be varied as it moves in and out of cities and intersections in the city can have different settings than those in the country. Lowering the aggressiveness in the city will limit the cause of rollbacks while lowering the risk will limit the extent of rollbacks. Increased aggressiveness and risk on the highway will take advantage of the available parallelism. It is clear that the Chandy-Misra system will not perform well for this model. Time Warp models would have to be modified to allow the cascading rollbacks to be limited and the efficiency improved. Unfortunately, this modification is non-trivial and would be at the expense of model clarity. The performance of Time Warp could also be improved through the use of a global multiprocessor load balancing scheme. This would ensure that at a given point in time, only the components with the

least LVTs will be simulated by all processors. 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. Many of these problems can be overcome in this specific case but not in general.

The traffic system example is simplistic but it illustrates the usefulness of variable synchronism. A simple version of the traffic system has been implemented in Rival and is detailed in Appendix A. Appendix A also contains the results of some preliminary performance analysis done using the traffic system example. Although this testing did show that performance increased when risk and aggressiveness values were between 0 and ∞, the test suite was quite limited. As such, we cannot make any conclusive statements regarding the performance of Rival.

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). Our initial runs of the example indicate that varying the risk and aggressiveness can have a substantial effect on the behaviour of the simulation (e.g., rollbacks). This indicates that the UDS algorithm may be capable of extracting significant parallelism from system models. The interested reader is referred to Appendix A for more details on the test and its results.

Summary of implementation

The current prototype implementation of Rival demonstrates the concept of Unified Distributed Simulation. Rival has not had a chance to grow and mature, consequently, it lacks user tools for building simulations. Without tools for managing

and manipulating concurrent tasks, it is difficult to coordinate the startup of a simulation. Special tools are also required for monitoring simulation progress and gathering output and statistics.

One such tool is a graphical representation of the LVTs of the models which shows how far into the future components are projecting. Such a representation might look something like figure 4.9 where the vertical axis represents time with the current GVT defining the origin. Each bar shows the difference between a component's LVT and the GVT. Using a tool like this, users can see which components are holding back the simulation and which are rolling back often. They can then manually vary the risk and aggressiveness of these objects and observe the changes in system behaviour.

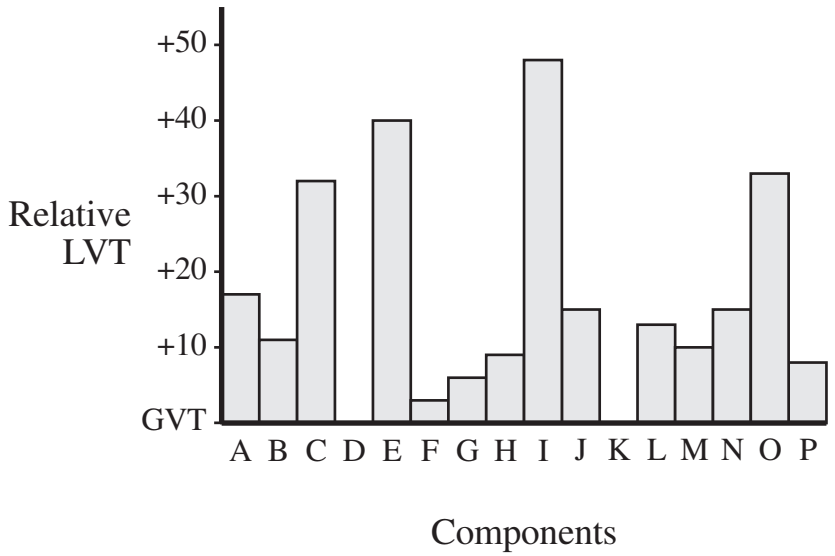

Figure 4.9 A graphical LVT display

We hope that further work will lead to a more robust implementation of the simulation mechanism and the application of UDS concepts to general distributed

computing problems. For example, it may be possible to build a distributed scheduler based on variable risk and aggressiveness which uses reflection to automatically adjust task behaviour.

Chapter 5: Conclusions

We have presented Unified Distributed Simulation, a technique 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 has not been conducted, the experiments presented in Appendix A indicate that the UDS algorithm may be more efficient for some systems. Additionally, 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 pessimistic 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 *futures* promises to reduce the cost of speculation 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.

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, pp. 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), pp. 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*, The Second International Conference on Computers and Applications, Beijing, China, pp. 694-698, June.

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

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

Briot, J-P and Cointe, P. (1987) *A Uniform Model for Object-Oriented Languages Using the Class Abstraction*, IJCAI '87, Milan, Italy, pp. 40-43, August.

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

Carriero, N. and Gelernter, D. (1989) *Linda in Context*, Communications of the ACM, 32(4), pp. 444-458, April.

Chandy, K.M., Holmes, V. and Misra, J. (1979) *Distributed simulation of networks*, Computer Networks 3, North-Holland Publishing, pp. 105-113.

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, pp. 440-452.

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

Cointe, P. (1987) *Metaclasses are First Class : the ObjVlisp Model*, Proceedings of OOPSLA '87, Orlando, FL, pp. 156-167, October.

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, pp. 229-244, May.

Gafni, A. (1988) *Rollback mechanisms for optimistic distributed simulation systems*, Proceedings of Distributed Simulation 1988, San Diego, CA, pp. 61-67, February.

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*, Distributed Simulation 1988, San Diego, CA, pp. 45-49, February.

Goldberg, A. and Robson, D. (1983) *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley.

Halstead, R.H. (1986a) *An Assessment of Multilisp: Lessons from Experience*, International Journal of Parallel Programming, 15(6), pp. 459-501, June.

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

Haynes, C.T. (1984) *Engines Build Process Abstractions*, Proceedings of the 1984 Symposium on Lisp and Functional Programming, Austin, TX, pp. 18-24, August.

Jefferson, D.R. and Sowizral, H. (1985) *Fast Concurrent Simulation Using the Time Warp Mechanism*, Proceedings of the Conference on Distributed Simulation 1985, San Diego, pp. 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), pp. 77-93.

Kaudel, F. (1987) *A Literature Survey on Distributed Discrete Event Simulation*, Simuletter, 18(2) pp. 11-21, June.

Kranz, D.A., Halstead, R.H. and Mohr, E. (1989) *Mul-T: A High-Performance Parallel Lisp*, Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, Portland, OR, pp. 81-90, June.

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

Lamport, L. (1978) *Time, Clocks, and the Ordering of Events in a Distributed System*, Communications of the ACM, 21(7), July.

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, pp. 50-55, February.

Lubachevsky, B.D. (1988) *Bounded lag distributed discrete event simulation*, Proceedings of Distributed Simulation 1988, San Diego, CA, pp. 183-191, February.

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

Maes, P. (1987) *Concepts and Experiments in Computational Reflection*, Proceedings of OOPSLA '87, Orlando, FL, pp. 147-155, October.

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

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

Misra, J. (1983) *Detecting termination of distributed computations using markers*, Proceedings of the 2nd ACM Principles of Distributed Computing, Montreal, pp. 290-293.

Misra, J. (1986) *Distributed Discrete-Event Simulation*, Computing Surveys, 18(1), pp. 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, pp. 44-56.

Reynolds, P.F. (1988) *A spectrum of options for parallel simulation*, Proceedings of the 1988 Winter Simulation Conference, San Diego, CA, pp. 325-332, December.

Samadi, B. (1985) *Distributed Simulation, Algorithms and Performance Analysis*, Ph.D. dissertation, University of California Los Angeles, Los Angeles, CA, January 1985.

Sokol, L.M., Briscoe, D.P. and Wieland, A.P. (1988) *MTW: A strategy for scheduling discrete simulation events for concurrent execution*, Proceedings of Distributed Simulation 1988, San Diego, CA, pp. 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*, Proceedings of OOPSLA '88, San Diego, CA, pp. 306-315, September.

Yonezawa, A., Briot, J-P., and Shibayama, E. (1986) *Object-Oriented Concurrent Programming in ABCL/1*, Proceedings of OOPSLA '86, pp. 258-268, September.

Xerox, Smalltalk-80 is a trademark of Xerox Inc.

Appendix A: The traffic example

We have implemented a simple version of the traffic example described in Chapter 4. The example contains Intersections and Cars. Figure A.1 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 the associated edge.

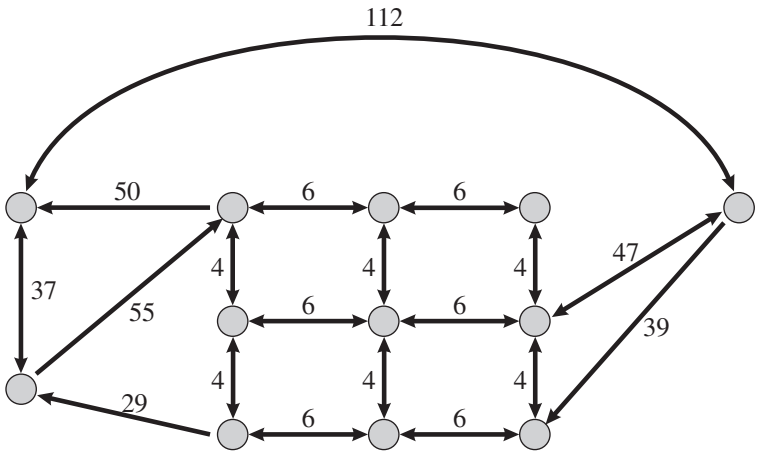

Figure A.1 Example road network

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 an edge as the argument. The car will proceed along the link, 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 A.2. 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 A.2 Source code for traffic system example

A number of test runs were done with this example and the results are given in figure A.3. The system was run on two processors, one approximately 30% faster than

the other. Even though each processor ran an equal number of like simulation components, it was predicted that the highly connected intersections would be the busiest and so they were placed on the faster processor. In each run, the simulation was executed until a particular simulation time was reached. The number of real seconds required to reach this point was recorded.

		Risk and Aggressiveness		
		0	20	∞
Cars	6	235	90	180
	12	470	185	300

Figure A.3 Traffic system results

We can see from figure A.3 that a moderate risk and aggressiveness can significantly improve performance. There are a great many variables which will influence the results of such experiments. The first is task distribution. Which tasks are put on which processors and the behaviour of those tasks will affect processor loading and thus performance. The cost of interprocess and interprocessor message passing will also affect performance. If the cost is low then the cost of rollbacks will be reduced. If there is a significant difference between the cost of interprocess and interprocessor messages then task distribution and interaction becomes more important. Since the cars have non-deterministic behaviour, any change in the running of the simulation may affect the behaviour of the cars. Of course, the risk and aggressiveness settings and the number of processors used will also have an effect.

Also, the optimal value for the risk and aggressiveness will depend on the time scales of the system being simulated. In the traffic system the interaction times are quite low so small amounts of risk and aggressiveness can be effective.

It should be noted that Rival is not a particularly efficient implementation of the UDS algorithm. Further, efficiency of the deadlock recovery scheme (i.e., the GVTEstimator) has not been established. This may have significant bearing on the results when low values are used for the risk and aggressiveness. We feel that Rival does represent a reasonably good implementation of the Time Warp algorithm however. In summary, this experiment provides strong evidence that the Unified Distributed Simulation Algorithm can improve the performance of some distributed simuations.