4. A Universal Simulation System

Chapter 4

A Universal Simulation System

"Everything should be made as simple as possible, but no simpler."

Albert Einstein

Given a solution for distributing a VE at any level (near/tightly-coupled to far/loosely-coupled), it is next to impossible for it to be successfully applied to the other levels without, at best, some loss in efficiency and, at worst, complete failure to meet the system requirements. DIS, SIMNET, DIVE and MR Toolkit are all designed to operate at one level and hence do not scale well. AVIARY has a more flexible design but little thought has been given to large-scale distribution. WAVES has been specifically targeted at low-end systems and has resulted in an architecture designed to compensate for a low bandwidth.

The remainder of this thesis presents a system architecture which fits the many different combinations of computational power and bandwidth that may be found in networked simulation systems. This is not done by applying one solution at all levels, but by a number of solutions each best applied to a certain level, all of which share an underlying structure and philosophy. Deciding how the architecture is applied to a particular configuration will be the responsibility of the system designer/administrator. Enforcing a strict organisation would present problems considering the diversity of hardware that may be used. It is possible, however, to derive a set of guidelines which can be used to aid this decision process.

There are some tasks that do not distribute well. Take, for example, image generation which, if it is to be distributed at all, must be done over a tightly-coupled network (with current technology) due to the high update rate that is necessary and the large volume of data that is generated. Similar arguments may be made for acoustic rendering and other local phenomena. It is no coincidence that these tasks are all to do with input and output. As section 3.3 showed, a high fidelity VE can be made or broken through the participant's view of the environment. Introducing lags and hence loss of fidelity by distributing these tasks will work against the intended goal. On the other hand, rendering the environment (let alone simulating it) can be a large computational burden. Therefore there is a clear need for local distribution of the simulation so that larger computational resources may be accessed whilst maintaining the fidelity of the simulation.

Some tasks, however, do distribute well. In fact their distribution is the key to their success such as a simulation with a very large number of entities. For example, 100,000 entities and

upwards cannot possibly be simulated locally (with reasonable expense) and requires a larger set of resources to complete the task. The ability for simulations to operate over large distances is a natural progression and applications are easy to foresee, but the implications of such geographically dispersed distribution are many and substantial. The delays introduced by bandwidth limitations, switching stations, routers and protocol overheads can severely affect interactivity.

This chapter presents the design for the Universal Simulation System (USS). First of all, the system requirements are described, followed by a summary of some design restrictions with regards to real-time and distributed systems in general. Before describing the system components that constitute the USS, the Universal Modeling Language (UML) is presented: a representation of the abstract model presented in chapter 3. If a USS is likened to a house, the system's components are the bricks and the modeling language is the cement that binds them and permits them to function together. The reader should note that use of the term "Universal" reflects the abstract model around which the VE is structured, i.e. our Universe. Its use is not intended to convey the impression of a solution that may be used for all types of simulations/modeling tasks.

4.1 System Requirements

Before proceeding further, let us first state the requirements that must be fulfilled by the USS:

1. *Real-time constraints*. The simulation must maintain a level of integrity that matches its application. For example, a simulation which must support human interaction, e.g. a driving simulator, must provide a high, constant environmental update rate. When modeling a complex system that exceeds the computational limits of the hardware, much lower constraints may be set that, although not interactive, must still be met.

2. *Scaleable from small to large scale simulations*. It should be possible to take the same simulation model and distribute it at all levels with the minimum of effort, preferably transparently.

3. *Multiple human participants*. Man-in-the-loop simulations introduce new restrictions on the simulation system, e.g. large lags are unacceptable. Multiple people interacting within the same VE increases the complexity of executing the simulation proportionally.

4. *Applicable to a wide range of simulation applications*. Rather than concentrate on one class of simulation, the system should provide sufficient generality in its structure such that it may be applied to many different types of simulation.

5. *Flexible distribution*. There should be no enforced structure for distributing the simulation and the resources. The system should adapt around the simulation and not *vice versa*.

6. *Resource optimisation*. To maximise the use of available resources the simulation workload must be capable of being redistributed where possible.

7. *Fault tolerant*. A minimum of degree-3 fault tolerance should be supported with as little impact on performance as possible.

8. *Secure*. Steps must be taken to ensure that each system component cannot be violated thus compromising system security.

4.2 Design Restrictions

There are also several limiting factors that must be addressed when considering system solutions.

4.2.1 Finite Memory

Whether we talk in terms of physical memory or virtual memory there is still a finite amount that can be used before performance suffers. At the time of writing, a typical IBM PC has on average 4-8 Mbytes of memory. This is often increased for specialist applications such as 3D modeling but this is uncommon. By contrast, a middle-range SGI Onyx will come with 64 Mbytes as standard. Memory is often the most expensive component of any system and therefore physical memory should be seen as a precious resource.

4.2.2 Finite Computational Power

Some of the systems reviewed used total replication of the VE on each node as a solution to some of the issues presented. However, a node can only process so much and that limit may easily be exceeded when simulating larger VEs. Excessive demands can be placed on the CPU if it also has to process network packets. On faster networks or in a large simulation, this can become a bottleneck when the CPU fails to keep up with the traffic. This problem may be alleviated if the node has the luxury of multiple processors but such systems are more expensive and thus less common.

Even if a CPU was dedicated to communications it would be a wasteful use of resources if a change in protocol resulted in much lower traffic. If this was the case then it would permit the savings to be applied to the simulation. Regardless, maximum use should be made of all available computational resources, whether they are on the same node or over a network.

4.2.3 Finite Communications Bandwidth

Table 4.1 shows a summary of the more popular systems for forming networks, their target network class, bandwidth and the physical medium used for connecting the nodes. At first glance it might seem that the larger the geographical distance between nodes, the higher the bandwidth available to the node. This is a false picture because the number of nodes connected typically increases as we move from LAN through to WAN technology. Therefore, in general, the longer the distance covered by a network, the smaller the effective bandwidth available to each node. If a VE system designed for a LAN saturates the bandwidth then this in itself will be enough to cause problems when it is expanded to cover a larger geographical area.

System	Theoretical Bandwidth	Class	Medium†
V.34 Modem	28.8 Kbps	Dedicated link	Copper telephone line
ISDN	64 Kbps per channel	Dedicated link	Copper telephone line
Frame Relay	56 Kbps - 1.98 Mbps	WAN	Coaxial
Ethernet	10 Mbps	LAN	Coaxial or twisted-pair
Fast-Ethernet	100 Mbps	LAN	Coaxial or twisted-pair
CDDI	100 Mbps	LAN	Twisted-pair
FDDI	100 Mbps	LAN - MAN	Fibre-optic
ATM	155 Mbps+	LAN - WAN	Fibre-optic

LAN	Local Area Network	ATM	Asynchronous Transfer Mode
MAN	Metropolitan Area Network	CDDI	Copper Distributed Data Interface
WAN	Wide Area Network	FDDI	Fibre Distributed Data Interface
		ISDN	Integrated Digital Service Network

†This is the medium used to connect the node, it does not reflect the national backbone which would likely be fibre-optic.

Table 4.1 Networking medium properties.

The actual bandwidth available will vary depending on the protocol used across these mediums and the amount of traffic, with the exception of Frame Relay and ATM. These two systems permit channels of a specified bandwidth to be allocated and hence bandwidth is guaranteed during the existence of that channel.

4.2.4 Limited Transport Mechanisms

Since the architecture will be applied to diverse hardware/software platforms no assumptions may be made about the type of communications supported. Some may provide proprietary messaging systems, others may use TCP or UDP. Point-to-point communications are fairly standard although their implementation may not be readily conceptualised as message-passing: a multi-processor system may use shared memory and semaphores.

Broadcast facilities are quite specialised and dependent on the transport medium - multicast is even more rare. If these forms were available, the issue of reliability must still be dealt with.

4.3 Distributed Real-Time System Implications

A typical real-time system consists of many processes, each of which has a very specific task. Usually a process is dedicated to waiting for a specific event to occur, e.g. an interrupt, and then performs some work when it is triggered. There are two types of real-time systems: *soft* and *hard*. In a soft real-time system each process performs its work as fast as possible and if it misses its deadline for completion nothing catastrophic will happen. Hard real-time systems, on the other hand, require that each process must complete their work before the deadline. Exceeding the prescribed finish time is a system failure and can result in disastrous

consequences, e.g. the fly-by-wire systems found in high-performance aircraft have hard constraints.

4.3.1 Computation Management

Section 3.3 discussed a specific problem with current VE systems which may be placed in the soft real-time category. We shall therefore only concern ourselves with hard real-time systems. Cheng (1988) presents a review of the key scheduling algorithms and their application to distributed systems. A more detailed taxonomy can be found in Rotithor (1994) but Cheng's taxonomy will suffice for this section (Figure 4.1).

4.3.1.1 Static

Static scheduling relies on the knowledge that the number of tasks and their characteristics will not change at run-time. This permits off-line scheduling to be performed and tested until a suitable schedule is found. One such tool for this is generalised rate monotonic scheduling

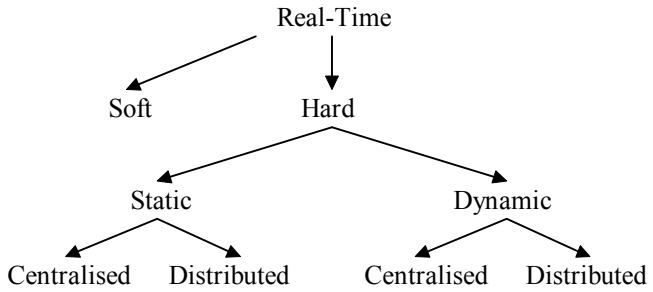


Figure 4.1 A taxonomy of real-time scheduling algorithms.

theory (Sha and Sathaye, 1995). The CPU is allocated to the highest-level priority process which preempts execution of lower-level priority processes when needed. Their priorities are fixed and changing them can be a costly process. This process is therefore usually undertaken at the system design stage or when considering changes to an established system

In a VE system, entities may be created and destroyed at run-time and the complexity of calculations performed may vary, e.g. collision detection. It is also possible that the communication paths between entities will not be static: depending on system design, each process may be able to communicate directly with each other. These three points defy the application of static scheduling.

4.3.1.2 Dynamic

The dynamic method schedules processes at run-time and permits more processes to be added to the schedule and others to be removed. Although dynamic schedulers incur higher overheads compared to static schedulers, they are the only type applicable to VE systems. A process may be characterised by its timing constraints, precedence constraints and its resource requirements. Timing constraints can be described by the four parameters:

- *Arrival Time*: the time at which the process is invoked in the system.

- *Ready Time*: the earliest time at which a process can be executed.
- *Worst Case Computation Time*:
  the execution time of the process is always less than this.
- *Deadline*: the time by which the process must finish.

Processes may be *periodic* or *non-periodic* (*aperiodic*). A periodic process executes once per time period whereas a non-periodic process executes only once and whose arrival time and deadline are unknown until run-time. In a simulation a large percentage of events will be the same for each time step, e.g. sending update messages, updating displays, etc. These events are periodic processes (although computational work may still vary) and the remaining unpredictable events likened to aperiodic processes.

Precedence constraints represent the order in which the processes must execute and may be described as an acyclic directed graph. This graph may change as new processes arrive. An added restriction is whether a given process is *preemptable* or *non-preemptable*. That is, can it be interrupted after it has started execution and resumed afterwards or must it run to completion unhindered?

The success of a dynamic scheduling policy can be measured by its *guarantee ratio* which is the total number of processes guaranteed to meet their deadline versus the total number of processes that arrive.

4.3.1.3 Centralised

Cheng's centralised classification refers to systems where the processors are tightly-coupled and the cost of Inter Process Communication, IPC, is negligible. A number of algorithms have been proposed to solve the problem of dynamic scheduling in a centralised system (Locke *et al.*, 1985), the most popular and proven of which is *earliest deadline first*. As the name suggests, the process that needs to finish next is executed first. A detailed evaluation of this algorithm can be found in Halang (1992).

4.3.1.4 Distributed

The distributed classification refers to systems which use loosely-coupled processors and IPC overheads can no longer be dismissed. Scheduling on one node is quite different from scheduling a distributed system. When this step is taken two fundamental changes take place:

1. All resource requests are no longer known to the centralised scheduler.

2. Communications latency means that events may be delayed and/or not appear in time.

The transmission delay must be incorporated into the process' schedule to ensure that its deadline is still valid. Also, the propagation delay may exceed transmission time on larger networks, so both must be accounted for.

Communications delays also mean that any central scheduling algorithm would be working on out-of-date information about each node. For this reason distributed systems usually have two scheduling components: a *local scheduling* algorithm and a *distributed scheduling* algorithm.

The local algorithm determines whether the process can be executed locally and, if not, the distributed component determines where in the system it should run. Centralised scheduling policies may be used as local algorithms but new solutions must be used for global scheduling.

When allocating a process to a node, the target may be selected either by choosing the node with the lowest load (*focused addressing*) or through a *bidding* process whereby each node bids for the task. The former uses out-of-date information but communication latency is low, whereas the latter uses accurate node information but incurs high communications latency. Stankovic *et al.* (1985) present an algorithm that combines both of these techniques, including the overheads due to scheduling and the communication delays between nodes. To reduce the amount of computation required to find an optimum schedule, heuristics and estimation techniques are used.

Many distributed systems employ *load-balancing* algorithms (Boutaba and Folliot, 1993; Gavish and Sridhar, 1994). However, these deal only with workload management and do not consider timing constraints. They may, therefore, be seen as a simple case of the general distributed scheduling problem.

Some algorithms are static in that once a process has been allocated to a node, it remains there for the duration of its execution. Dynamic algorithms impose no such restriction and permit a process to move from one node to another, a technique often called *process migration*. Naturally, there are reasons for moving a process which are not based merely on node loading. Migration may be used to great effect if a process begins to perform an intensive task over a network link that may be best performed local to the resources it needs[1]. For example, a process interrogating a large database of information stored on disk would take much longer and consume large amounts of communications bandwidth unless it was located on the node with the actual disk. Fault tolerance also provides an incentive for migration (section 4.3.5).

4.3.1.5 Service Degradation

Ensuring that the VE appears to be behaving correctly to the participants requires that all visible entities and dependent system processes meet their deadlines. In a large VE this may be a small subset of the total entities which opens the possibility of enhancing the scheduling. If a process was designed to provide different levels of accuracy, e.g. loss of calculation accuracy traded for speed, then the guarantee ratio could be increased by using lower accuracy on those processes that are currently less important to the success of the simulation.

4.3.2 Memory Management

This is of special concern to real-time systems because memory management can be a costly venture. Virtual Memory (VM) is not used in strict real-time systems because it introduces a certain amount of unpredictability into the system. VM also requires an often significant amount of disk space to be put aside to hold any internal data structures that are generated at run-time. This does, of course, permit the execution of large processes but the overheads

[1] If a process uses a resource intensively throughout its lifetime then it should be allocated to the node local to that resource from the start.

incurred usually degrade system performance too much. Whilst there are compromises, such as the use of overlays which the application has control over, they are rarely used because disk accesses must still be scheduled.

4.3.3 Locating Resources

In a distributed system it is necessary to provide a mechanism through which a process may locate a resource that it requires during execution. This resource may reside on the same node as the process or on another node in the system. The solution is a directory of service providers and their location. Any process may then interrogate the directory using, for example, the name of the service and retrieve the actual address of the service which is then used to communicate with them. Such *name servers* may be either integrated into the operating system kernel or run as separate processes (Bowman *et al.*, 1990). So that all system-wide location registrations are recorded they must be communicated to the name server. If this service is embedded in the operating system kernel then extra name server functionality must be added. A separate name server process does not increase kernel complexity whilst achieving the same end result.

If only one name server exists then it is a weak point in the system and its failure (or loss of communication with it) could render the system helpless. It is common, therefore, to enlist multiple name servers which keep each other informed of registrations (QNX, 1993). Apart from increasing fault tolerance, multiple servers also increase the service response time for registration and location requests.

4.3.4 Location of Backing Store

No assumptions about the location of backing storage can be made in a distributed system. Diskless workstations are still widely used where all programs and data must be sent back and forth along the network link to a central server complex. Typically the operating system makes this difference transparent to both the user and the applications by providing a local virtual filesystem (QNX, 1993). Therefore any system design should bear in mind that this resource may not be readily available. In addition, dependency on backing storage will slow any process down and increases scheduling complexity.

4.3.5 Fault Tolerance

The type of fault-tolerance required in a distributed system is influenced by the form of data and computation distribution employed (as discussed in section 2.4.6). Complete and partial distribution require full redundancy, i.e. a total duplication of the computation and/or the data. Failure to communicate with a given process must either result in communication with a backup copy of this process or waiting for access to that process to be restored. The same is true for partial data distribution. Partial computation replication inherently provides a certain degree of fault-tolerance because the high fidelity calculations are approximated on every node with interest in that process' work. Total replication, of course, already provides full redundancy.

Token fault tolerance may be achieved by duplicating the key system components such as the name server discussed in the previous section. To prevent such an approach having a large detrimental effect on performance it requires a low-overhead synchronisation method to keep each duplicate up-to-date. Such a suitable mechanism would be the use of multicast communications between duplicates.

Failure of all the hardware on one node is quite uncommon, a more realistic scenario would be for an individual hardware component to fail. If another functionally identical piece of hardware exists on another node then there is the possibility of moving the process dependent on this hardware to the other node. Process migration driven by hardware failure is a special case of the general load balancing task. In order to repair the hardware component it is possible that the node must first be powered down, e.g. replacing an integrated component rather than a device hooked up to an external I/O port. In this case all processes would have to be migrated to another node until the problem was fixed and then the current system load re-distributed. The same reasoning can also be extended to failure of key software components. Except in this case the faulty application could likely be fixed without taking the whole system down.

Unsuccessful attempts to communicate with a hardware or software component can be used as an indication of a fault. Alternatively, a failure may result in a partial or reduced quality service in which case it would be possible for a component to explicitly indicate failure.

4.3.6 Summary

A distributed real-time VE system is best equipped with a dynamic deadline scheduler. Most processes in such a system will be preemptable due to system call usage such as message passing. Two scheduling policies are best employed to work at different levels: local and global. The earliest deadline first algorithm provides a proven local scheduling policy whilst an effective global policy combines both dictation and volunteering techniques.

Memory is a finite resource and any design should treat it as such whilst system-wide resources may be brokered using a number of mirrored name servers. Access to any such resource, including backing store, must be carefully scheduled. Finally, the form of data and computation distribution used has a direct impact on the degree of fault-tolerance a system can support. The remainder of this chapter presents a system design driven by these observations, starting with the design of a modeling language.

4.4 A Universal Modeling Language

The UML is the *representation* of the *abstraction* of our universe. It is a description of the universe based on the framework defined in section 3.1.5, but imposing no restraints on what information should appear and where. Interpretation of UML and the subsequent execution of the model it represents provides us with an *implementation* of our model. Description of the system architecture would be impossible without referring to UML because it is integral to the system's design, thus it is presented first.

4.4.1 Language Requirements

Based on the analysis of modeling techniques in chapter 3, the design issues (chapter 2) and implications presented in this chapter, the requirements for UML are:

- Structure based around the abstract model of our universe.
- Easy data modeling.
- Easy to learn and familiar in structure.
- Fast incorporation of changes into the model.
- Portable across many hardware/software platforms to support process migration.
- Low resource overheads, e.g. memory, computation, etc.
- Co-operative with the implementation language.

Fortunately, the abstract model that has been proposed (also) strongly resembles that of an object-oriented model. The universe corresponds (using C++ terminology) to the class, the constants and properties to the member variables, the laws correspond roughly to member functions and entities would be the objects instanced from the class.

The remaining requirements make the choice of language a little more tricky. To enable easy modeling of the VE the language must be concise, unambiguous and high level. These criteria help narrow the search as does the requirement that the language is easily learnt and intuitive.

On a practical note, in order to promote use of the language, it should be accessible by as wide an audience as possible and hence procedural as opposed to functional. Whereas functional languages have been used for Virtual Reality "programming languages" (Coco, 1992), they are not widely accepted and are often difficult to read. An object-oriented based procedural language would therefore seem a fair compromise.

To aid in development, debugging and provide run-time flexibility, it should be possible to make changes to the representation at any time. The ability to add properties to an object (or remove them), redefine the laws governing the properties and possibly even changing the value of (the somewhat inaccurately named) constants is potentially immensely powerful. In theory, it could be possible for the complete simulation to be re-designed on-line. The implications of such an ability are mainly the concern of the implementation but it is evident that the language must have a clear structure and well-defined rules to minimise the confusion this could cause.

4.4.1.1 Compiled

Permitting the representation to change during run-time gives us two alternatives. Firstly, to use a compiled language that permits dynamic loading and secondly, an interpreter. Normally, a compiled language takes a number of compiled language files (object files) and links them together to produce one executable. Dynamic loading refers to the ability to take an object file and link it into the process' executable image whilst that process is running. Asides from the considerable problems preserving access to the program data, there are two problems with this solution. To create the object file a compiler must be used which can be quite expensive with regards to how much of the computer's resources it uses, e.g. a C++ compiler performs many optimisations and is often dependent on many header files, can

generate large temporary files, and so on. In fact the presence of local backing storage and sufficient memory to run a compiler is by no means certain. Secondly, the process of dynamic loading is operating system specific and is rarely done in the same way each time. Notably, under real-time operating systems dynamic loading is not available at all since it is undeterministic and hence undesirable. As far as process migration is concerned, some additional mechanism must be devised such that the modified code may be transmitted to the destination machine.

4.4.1.2 Interpreted

The second alternative is not without its negative points either. An interpreted language is often slow to execute in comparison to the fast-as-possible execution of a precompiled language. It is slow because the program is usually translated into an internal code, in which each instruction corresponds to a number of native machine code instructions. This weakness is also an interpreter's strength since the language is inherently portable across different architectures. If each machine was provided with a copy of the interpreter, the same program can run unchanged and execution speed may be improved by pre-translating frequently used routines (in a library for example) into the internal code which is then stored for later execution. Any further optimisation would require the coding of commonly used routines in the implementation language (IL) and compiled into the native machine code.

4.4.1.3 Resource Implications

If each entity is to be described using an interpreted language then it is essential that the amount of resources consumed by the interpreter is kept at a minimum. For example, in a simulation where hundreds or thousands of separate entities are being simulated, the overheads per entity soon become a real issue. Ideally, the language will be compact, concise and execute quickly. Unfortunately, this requirement conflicts with the ability to make modifications to the data description and the code at run-time. Such a flexible system will inevitably require more memory for the dynamic data structures and more processing time to administer them.

Even in the best case that we can hope for, the interpreted language will still run slower than a compiled language, or will take more memory or any number of other disadvantages. It is therefore desirable to code the frequently used or critical routines in the IL. In other words the interpreted language will be embedded and therefore some way of sharing code and data structures between the languages must be provided. It could be arranged such that the presence of compiled routines would override the interpreted definitions and hence this would not affect portability of the program, only speed.

4.4.2 Candidate Languages

The features we are therefore looking for in our potential candidates are:

- Interpreted.
- Procedural.
- Object-oriented (at least some form of inheritance).

- Extensible.
- Fast execution.
- Compact.
- Embedded.
- Available at no financial cost on many platforms.

Availability of the language at no cost on disparate platforms is essential and, if modifications are to be made, the source code is also required. A number of existing languages were evaluated to varying levels for their suitability: Bob, Glish, ICI, Lua, and Python. Other potential candidates were ruled out at an early stage due to lack of features, e.g. Application Executive (Bliss, 1991), or availability. Java (Gosling and McGilton, 1995) was released in late 1995 at which time software development for this thesis had ceased. Smalltalk is a financially expensive language that shares many features with Java such as supporting run-time code changes, but not run-time class structure changes. Since classes would be used to structure the model, this also rules out Smalltalk and Java as candidates.

4.4.2.1 Bob

Bob is an interpreter for a language with C-like syntax and a class system similar to C++, but without variable typing and mostly without declarations (Betz, 1991). All class data members are protected by default and may only be modified through a member function. Single inheritance is supported (not multiple) and Bob preserves the concept of constructors which may, unusually, initialise objects already in existence. Bob's interpreter takes the source code and compiles it so that it may be interpreted using a stack-oriented byte-code machine. This way, syntax analysis is performed only once (at compile time) and speeds up the execution considerably. With a little effort it is possible to extend the language to include more built-in types and routines written in the implementation language: C. The current implementation is written for MS-DOS but there is no reason why this language cannot be ported to other operating systems.

4.4.2.2 Glish

Glish is targeted at loosely-coupled distributed systems and the philosophy used is that individual programs in a system should be wholly modular, having no knowledge of other programs or data types that might exist (Paxson, 1993). Programs may communicate without knowing about each other through *events* which are name/value pairs. Glish has three main components: a scripting language for specifying what programs to run and how to interconnect them; a C++ class library so that programs can generate and receive events and manipulate data; an interpreter for executing the scripts. The language is array-oriented and is geared towards the manipulation of data sent between programs. By default all IPC is done through the interpreter which allows dynamic modification and re-routing of data but it is also possible to establish point-to-point links when performance is critical. Glish is written in C++, uses TCP/IP for its IPC mechanism and is available on SunOS, Ultrix and other UNIX variants.

4.4.2.3 ICI

ICI is an interpreted procedural language that represents C with extensions for built-in handling of arrays, structures and sets (Long, 1992). Structures are a key element of ICI, especially the notion of super structures (analogous to parent classes). If a reference to a member of a structure cannot be resolved then a search is made of that structure's super structure (if it has one). If the super structure does not contain the reference then the search proceeds to its super structure and so on. Although ICI is not object-oriented this mechanism provides a method for supporting inheritance albeit for data only (functions may not be members of structures). New data structures and functions may be defined at run-time but existing structures or functions cannot be modified.

4.4.2.4 Lua

Designed to be used for extending applications, Lua is a procedural language that makes heavy use of associative arrays that may be constructed and manipulated in many different ways (de Figueiredo *et al.*, 1994a, 1994b). Unlike ICI, Lua distinguishes the functions and data provided by the host application from the data and functions defined in the language itself. The other built-in types are strings, floating-point numbers and nil - the type of the *nil* variable. Only a small number of built-in functions are provided but embedding C routines from Lua is easily done and the Lua program may be extended at runtime. The language itself has very few constructs yet proves to be quite expressive. Rather uniquely, persistence of data may be provided by writing Lua code that writes Lua code that, when executed, restores the values of all global variables. Using a byte-code interpreter similar to the one in Bob, it is feasible to pre-compile the programs into byte-code form to decrease loading time and reduce runtime support.

4.4.2.5 Python

The designer of Python describes it as "*... a simple, yet powerful programming language that bridges the gap between C and shell programming, ...*" which is a very fair evaluation (van Rossum, 1994c). Python is rich with the familiar procedural programming constructs, provides exception handling as standard and comes with a large number of modules which provide interfaces to library routines varying from POSIX system calls to Silicon Graphics GL (van Rossum, 1994b). Modules have generally been pre-compiled, which can also be done to user code. A class mechanism has been added to the language since conception (with little trouble) and supports member variables, functions and multiple inheritance. Writing C functions and using them from Python is not an easy task, most of the complexity is to due to the memory management system used. To its credit, Python is the only language to support dynamic loading of extension modules (van Rossum, 1994a). By only loading a module when it is needed the core interpreter can be reduced in size and overheads. Unfortunately dynamic loading is currently only supported on some UNIX systems.

Language	Memory (Kbytes)	Memory + Program (Kbytes)	CPU Time (ms) ‡
Bob	224	340§	7,197
ICI	392	504	13,464
Lua	264	264†	13,653
Python	760	856	13,658
Compiled C	n/a	112	100

§ The implementation used had a memory leak which made accurate measurement impossible (this is a "best guess").
† Stack, heap and code space is statically allocated when the interpreter is compiled.
‡ 486DX 33MHz IBM PC Compatible running the QNX operating system.

Table 4.2 Interpreter resource evaluation.

4.4.2.6 Interpreter Performance

Glish, although ideally suited for the task it was designed for, is not really suitable for the task at hand. Adding to the interpreted code at runtime is not possible as the package stands currently and it requires that all input/output is routed through the Glish interpreter - this is not desirable. Each of the remaining languages, ICI, Bob, Lua and Python, fulfil most of the requirements. To determine how they compare when memory and Central Processing Unit (CPU) usage is examined, a test benchmark was written in each of the languages and measurements taken. The chosen benchmark was intended to test the speed of the interpreter with a typical task that would be easily represented in each language and not rely on the speed of built-in functions. The task was to multiply a four-by-four matrix with a vector 10,000 times (so as to average out the effects of variable lags in the operating system). In Bob and Python, the matrix/vector data types and manipulation functions were coded as a class, in Lua and ICI they were implemented as an Abstract Data Type (ADT) using normal functions and the relevant data structures. The amount of memory used by the interpreter (with and without the loaded program) and the CPU usage were measured.

On the whole there are little surprises in the results shown in Table 4.2. Both Bob and Lua offer few features and hence the interpreter is relatively small in size. ICI provides more elaborate data structures and language constructs and Python weighs in highest, not surprisingly due to its comprehensive range of features. Each interpreter takes about the same amount of memory to hold the program (~120 Kbytes) with the exception of Lua which has a fixed amount of space allocated when compiling the interpreter (this may, of course, be changed). The execution times are interesting in that all but Bob's time are almost exactly the same. These times do not include parsing overheads and so the efficiency of Bob's byte-code interpreter must explain its result. The figures for the same test written in C and compiled into machine code provide a good indication of how much time each interpreter really spends executing their translated code. The large differences in execution speed between machine code and interpreted code are not surprising, the machine code is optimised for the CPU in question and, for this particular example, so as not to stall the CPU pipeline and thus maximise throughput. Whereas the interpreters have to work through a considerable number

of other instructions in between each language instruction, making effective optimisation next to impossible.

4.4.2.7 Language Selection

Considering that there will be many processes in the VE system that will make use of the UML and hence the interpreter, the amount of memory used is a prime concern. Multiple copies of the interpreter itself may be avoided by putting it into a shared library which will only be loaded once. The amount of memory used to store the program is still an issue however, as is the amount of CPU consumed. If the language is overly complex it can lead to increased memory for storage and longer execution times.

Python suffers from difficult embedding and whilst, in an ideal world, its rich language would be very useful, the author believes that for the purposes of this thesis the overheads are too large. Since this decision was made, Python has been chosen as the programming language for the Alice rapid-prototyping system (UVa, 1995). Alice runs exclusively on SGI machines which offer greater CPU power, more memory and larger disk storage than is available on the average workstation. Also, Alice does not have the same requirements as detailed in section 4.4.1.

ICI provides little more than multiple inheritance in the way of object-oriented features and execution times are too high. Lua is impressive but has no object-oriented features whatsoever. Bob is the most promising of the group, it has a small, useful set of features but lacks a robust implementation and documentation.

One of the main requirements was the ability to modify code and data structures at run-time. None of the languages reviewed enable the data structure to be altered on-line and strictly speaking, none support code modification either. However, it is conceivable that those languages supporting dynamic loading might permit the replacement of previously loaded modules. Even then, this would be a heavy-handed approach and relatively very slow.

There is also the issue of transforming the model into code. Using a general-purpose language will unavoidably involve using different terminology and possibly a structure sufficiently different to cause confusion. To ensure an easy transition from model abstraction to representation whilst reducing resource overheads, the author believes that a special, optimised language needs to be derived, learning from the languages reviewed.

4.4.3 Proposed Language

The features of the surveyed languages that should be kept are:

- Simplicity of expression.
- Compactness - both interpreter and intermediate code size.
- Classes and inheritance.
- Implementation language interface for embedding.
- Modules.
- Use of byte-codes and a byte-code machine for language execution.

The negative aspects that will not be used are:

- Inability to alter structure of data and code at run-time.
- Lack of code/data persistence.
- Type-less variables/parameters.

The Universal Modeling Language is a procedural language and possesses some object-oriented properties, notably inheritance and operator overloading. Multiple inheritance is not supported primarily because it complicates interpreter design[2] (for a discussion on this topic see Swawe, 1989; Bretthauer *et al.*, 1989). Its appearance is a mix between C/C++ and Pascal. Some of the expression notation has been taken from C++ to aid brevity whilst Pascal lends us clarity of description.

UML can be split into two halves. The statements that describe the data - its structure and content - and the code that manipulates that data. The actual language statements used to represent these two components are almost completely unique to each component. In other words, a UML description can be separated into two categories: data definition and instruction code.

4.4.3.1 Data Definition

There are two structural components: the `UNIVERSE` and the `ELEMENT`. These are used together to form a hierarchical framework within which the other components may be placed: `CONSTANT`s, `PROPERTY`s, `CONVERTER`s, `FUNCTION`s and other `ELEMENT`s. A simple grammar representing the basic relationships between these components is given in Figure 4.2.

```
universe  : UNIVERSE name { components }
          ;

components      : components component
          | component
          ;

component : constant
          | element
          | converter
          | property
          | function
          ;

element   : ELEMENT name { components }
          ;
```

Figure 4.2 Backus-Naur Form description showing relationships between UML components.

4.4.3.1.1 Universe

The starting point of a representation is the definition of the universe which is assigned a name for reference purposes (Figure 4.3). Within the universe, properties may be defined and

[2] That is not to say that it would be inefficient (Templ, 1993).

grouped into elements for convenience, functions may be written to act on the properties of the universe and hence provide a behaviour. The state of a universe is made up of entity instances (section 4.4.3.1.8). Three functions are mandatory for each entity: `Construct`, `Destruct` and `Update`. `Construct` is called when an entity is created (this is after all only a *declaration*, not an *instance*) and is typically used to give initial values to the entity's properties. The `Destruct` function is called when the instance is being deleted and may be used to perform any last actions. The simulation is progressed through a series of discrete steps, each one beginning with the execution of the `Update` function. It is also used as a focus for the synchronisation of the data within the simulation.

```
UNIVERSE Base
{
     ELEMENT Models
     {
          ELEMENT     Visual;
          ELEMENT     Aural;
          ELEMENT     Tactile;

          PROPERTY    visual      : Visual;
          PROPERTY    aural       : Aural;
          PROPERTY    tactile     : Tactile;
     }

     PROPERTY    models      : Models;
     PROPERTY    position    : REAL[3];
     FUNCTION    time        : REAL;

     VFUNCTION   Construct;
     VFUNCTION   Destruct;
     VFUNCTION   Update;
}
```

Figure 4.3 Example top-level UML description.

4.4.3.1.2 Types and Constants

Other components of the language will appear familiar, such as the built-in primitive types: `REAL`, `INTEGER`, `STRING`, and `BOOLEAN`. Classical "user-defined" types are in fact supported through elements.

The only data structure primitive is the list, which may be created from any type, built-in or element. If a dimension is given when defined then the size of the list is fixed and may not be changed at run-time. If no dimension is given, i.e. an empty pair of square brackets, then the list may grow and shrink. Therefore, a fixed list may be likened to an array and a variable list compared to a linked-list.

Constants may be declared at any level of scope within the universe but may only use built-in types.

4.4.3.1.3 Elements

While it is possible to embed the definition of elements and functions within the universe section, it can soon reduce readability as the number of properties and functions increases. It

is therefore possible to merely give a stub declaration and provide a full definition later on. UML does not require that certain definitions are placed in specific files and as such, any completion of a stub declaration must qualify which stub it is satisfying. In the example above, the `Visual` element was defined as a stub in the universe called `Base`. A possible full definition is given in Figure 4.4 with the name of the element reflecting its origin. This "dot" notation is used in any situation where a stub and a full definition need to be associated, i.e. elements, functions, etc. It may also be used by the other component types, e.g. properties, to modify definitions when using the interpreter directives (section 4.4.3.1.9).

```
ELEMENT Base.Models.Visual
{
     ELEMENT Colour
     {
           PROPERTY    components  : REAL[3];

           FUNCTION    Set( triplet : REAL[3] );
           FUNCTION    Get( triplet : REAL[3] );
     }

     ELEMENT Surface
     {
           ELEMENT Polygon
           {
                 ELEMENT Vertex
                 {
                       PROPERTY    coord : Vector;
                 }

                 PROPERTY    vertexList  : Vertex[];
                 PROPERTY    colour      : Colour;
           }

           PROPERTY    polygonList : Polygon[];
     }

     PROPERTY    surfaceList : Surface[];

     FUNCTION    Read( filename : STRING ) : BOOLEAN;
     FUNCTION    Write( filename : STRING ) : BOOLEAN;
}
```

Figure 4.4 A possible definition for the `Visual` element.

The `Visual` element contains two further elements, one of which defines an element called `Colour`. At this point no data is held within the `Visual` element since the colour element is only a declaration. The `Surface` element has further elements nested within it - there is no limit to the level of nesting permitted. The `Vertex` element declares two instances of previously defined elements: `Vertex` (local to `Polygon`) and `Colour` (local to `Visual`). Similar definitions may be made for `Aural` and `Tactile`.

Elements may be treated similarly to classes in object-oriented languages - they can define data and code which operates on that data. Even if the element does not define any properties, the element must be instanced before the element's functions may be called.

4.4.3.1.4 Properties

A property is formed by two parts, the name of the instance and a description of its structure separated by a colon. The property's structure may be based on a built-in type or an element. Only elements that have already been declared may be used in property declarations. A property declaration is an indication that the structure defined by an element or type should be instanced and hence take physical form.

4.4.3.1.5 Functions

A function is identified by its name (using dot notation if necessary), the parameters it requires (if any) and a possible return type. All parameters referring to variables and properties are passed by reference whilst literals are passed by value. By default a function does not have a return type. The contents of the function are made up of one or more imperative statements. A pure virtual function may also be declared using the `VFUNCTION` keyword, which means that no definition is provided at that level but must be provided by any universe inherited from this base universe. The `Construct`, `Destruct` and `Update` functions in this example are all virtual functions because the values of the properties are different for each instance, to provide default values only to be overridden by derived functions would be wasteful. In Figure 4.4 two functions are defined within the `Visual` element to input and output visual representations.

```
UNIVERSE PBM : Base
{
     CONSTANT    Gravity     : REAL[3] = [ 0.0, -10.0, 0.0 ];

     PROPERTY    mass        : REAL;
     PROPERTY    velocity    : REAL[3];

     FUNCTION    Construct
     {
           // Assign initial values for the inherited
           // properties.

           position = [ 0.0, 0.0, 0.0 ];


           // Now assign values for the local properties.

           mass = 0.0;
           velocity = [ 0.0, 0.0, 0.0 ];
     }

     FUNCTION    Update      { ... }
     FUNCTION    Destruct    { ... }
}
```

Figure 4.5 Defining a `UNIVERSE` by inheritance.

4.4.3.1.6 Inheritance

Inheritance is used heavily within UML to specialise descriptions of the universe. The example in Figure 4.5 shows that the universe `PBM` (Physically-Based Model) is derived from `Base`. In addition to all the properties, elements, constants and functions defined in `Base`, the new universe defines extra properties and provides a definition for the virtual functions.

Inheritance is not limited to universe components, elements can also be derived from other elements providing that they have already been declared. The parent element could be in the same scope level or even in an ancestor universe.

```
ELEMENT RGBColour : Colour
{
     CONVERT HLSColour { ... }     // Convert from RGB to HLS
}

ELEMENT HLSColour : Colour
{
     CONVERT RGBColour { ... }     // Convert from HLS to RGB
}
```

Figure 4.6 Inheriting from an element.

4.4.3.1.7 Converters

With the effective proliferation of a large number of elements (essentially types) it is often necessary to convert between one and another. In some cases this may be trivial, e.g. converting a string into a real, an integer into a real, etc. In other cases the transition may be less straight forward, e.g. converting from one colour model to another (Figure 4.6), changing a surface model description into a volumetric description, etc. To handle these non-trivial conversions special functions may be defined within an element that identify the result of the conversion by giving the destination type as their function name. Converters do not take parameters and do not return any value. They may be implicitly invoked by the interpreter or explicitly by the programmer as shown in Figure 4.7.

```
FUNCTION Colours
{
     PROPERTY    rgb   : RGBColour;
     PROPERTY    hls   : HLSColour;

     rgb.Set( 1.0, 0.0, 0.0 );     // Bright Red!
     hls = rgb;                    // Interpreter invokes
                                   // correct conversion
                                   // function.
     hls = HLSColour( rgb );       // Force conversion.
}
```

Figure 4.7 Explicit/implicit invocation of a converter.

In the event that a converter could not be found, an exception would be raised during interpretation.

4.4.3.1.8 Entities

The entities are the physical embodiment of the universe. An entity is created by specifying the universe in which it belongs and from this information it is furnished with a copy of the properties, elements, constants and functions defined for that universe. The `Construct` function is then called to initialise the entity's state. Some of this initialisation code may be

found in the universe definition but usually this is appended to, if not completely specified, in the entity definition. When an entity is destroyed its `Destruct` function is also called.

```
ENTITY Ball : PBM
{
     FUNCTION Construct
     {
          mass = 10.0;                    // kg
          velocity = [0.0, 1.0, 0.0];     // 1 m/s upwards
          position = [0.0, 10.0, 0.0 ]; // 10m straight up

          // Initialise models...
     }

     FUNCTION Update
     {
          VAR   force : REAL[3];

          force       = Gravity / time();
          velocity    = velocity + force;
          position    = position + velocity;
     }
}
```

Figure 4.8 Definition of an entity.

In the example shown in Figure 4.8 the `Construct` function overrides the default values that were assigned in the `Construct` function of the `PBM` universe definition given earlier. The `Update` function represents the actions to be taken at each simulation step, thus defining the entity's behaviour. In this case the universal function `time` (defined in `Base`) is used as the basis for a calculation to determine the entity's position after gravity has played its part.

Entities may also declare their own functions locally without requiring a stub declaration in the universe they are derived from.

4.4.3.1.9 Interpreter Directives

An interpreter directive is a special command which may be inserted anywhere in the definition and affects what the interpreter does with the following statements. There are currently only three directives which change the interpreter's mode of operation: *insert*, *replace* and *delete*.

Insert mode will add the component definition providing that a component with that name in that level of scope does not already exist. If it does exist then the operation fails. In replace mode the definition is always added, even when there is already a component with the same name. In this case, the old definition is removed and the new one inserted. When in delete mode the interpreter only uses the name of the component in order to locate it in the definition and remove it. If the component does not exist then the operation fails and an exception is thrown. The dot notation is used when specifying the component names so that they may be used to place/locate the component correctly.

4.4.3.2 Instruction Code

It was decided early in the design process that the instruction code aspect of UML would not be implemented (section 5.5.5). Hence only unique features and those that have an impact on the interpreter design and implementation are presented here.

4.4.3.2.1 Local Variables

Variables may be declared at the element and function scope level or any level of scope therein. The `Update` function in Figure 4.8 has a local variable which will be instanced each time the function is called, unlike property definitions which are instanced permanently for a given entity. Variables may be declared as a built-in type or an instance of an element defined within the universe it is derived from. In fact, a variable declaration is identical to that of a property with one exception: variables may be initialised on declaration with an expression as shown in Figure 4.12. Properties may only be initialised with a literal or list of literals.

```
ELEMENT Outer
{
     ELEMENT Inner
     {
           PROPERTY    number : REAL;
           PROPERTY    text   : STRING;
     }

     PROPERTY inner  : Inner;
     PROPERTY number : INTEGER;
}

FUNCTION Scope
{
     VAR   outer : Outer;

     outer.inner.number = 1.0;

     WITH outer.inner
     {
           number = 2.0;
           text   = "Hello World";
     }
}
```

Figure 4.9 Methods for accessing member properties in elements.

4.4.3.2.2 Element Referencing

When an element has been instanced, as either a property or a variable, then the contents may be accessed using the familiar dot notation as shown within function `Scope` in Figure 4.9. If the element has a large structure then referencing the contents can become tedious and clouds the expression of logic. UML provides a similar mechanism to Pascal by permitting a specified scope to be made temporarily local (using the `WITH` keyword) so the contents may be referenced as if they were declared locally.

If there should be a name clash when a scope is made local, such as that between the `number` property in the `Inner` element and the `number` property in the `Outer` element, then the former would be used. Multiple scopes may be processed by presenting them as a parameter list, each name separated by a comma.

4.4.3.2.3 Function Calls

Figure 4.10 shows a call to the function that reads data into a `Visual` element. The `Read` function only takes one parameter and returns a boolean value indicating success or failure. If the function should fail then a special system function is called which places a message onto the current output stream and an Input/Output exception is generated.

```
FUNCTION Construct
{
     // Initialise the visual model associated
     // with this entity.

     if ( models.visual.Read( "plane" ) == FALSE )
     {
          system.Print( "can't open file 'plane'" );
          throw EXCEP_IO;   // Fatal error.
     }

     // Rest of construction...
}
```

Figure 4.10 User and system function call execution.

4.4.3.2.4 Exceptions

Error handling is done almost completely by exceptions. They may be thrown by the interpreter when a severe error occurs or by user-defined routines that wish to pass control (and error resolution) back to a previous level of execution. If an exception handler does not exist around the call to the routine that generates it, then the next level is checked and so on back to the top level. Failure to catch an exception will eventually end in a fatal error and the interpreter will stop executing the UML description.

The code in Figure 4.11 manufactures an exception by attempting to convert a colour of type `RGBColour` to `HSVColour` when the latter provides no conversion function. A number of exceptions are predefined by UML, the conversion exception that is shown in the figure is one such example.

```
ELEMENT HSVColour : Colour
{
     // Definition without any converters...
}

try
{
     PROPERTY    rgb   : RGBColour;
     PROPERTY    hsv   : HSVColour;

     rgb.Set( 1.0, 0.0, 0.0 );           // Bright Red!
     hsv = rgb;
}
catch ( EXCEPT_CONVERTER )
{
     // Resolve problem.
}
```

Figure 4.11 Attempting to convert an element without a converter.

4.4.3.2.5 State Indexing

A state change occurs on completion of the Update function. It is possible that we may wish to reference old values of particular properties when performing the current state calculations. Figure 4.12 shows how the time difference between successive simulation steps may be derived. The number in the round brackets indicates which state should be accessed. A value of zero would be the current state and is implicit, -1 would indicate the previous state, -2 the state before that and so on.

```
FUNCTION Construct
{
     VAR   dt : REAL = time - time(-1);

     // Do something with dt...
}
```

Figure 4.12 Calculating a time delta using state indexing.

Obviously storing a history for each property would be grossly inefficient and unnecessary. It is for this reason that only literals may be used to reference states. When interpreting the code it is possible to identify those properties that need to be stored and the length of the history list. If variables were permitted to index states, the history list could be any length and would impose unattractive time and space overheads. If a number of states (only known at run-time) do need to be referenced then conventional methods can still be used, e.g. storing them in a list. In this example only one previous state needs to be kept for `time`.

4.4.3.2.6 Modules

Putting a complete universe definition in one file, complete with entity declarations, code, etc. is impractical. Splitting a program into modules is a common practice in other languages and this same technique is applied in UML. Each module is a file that contains syntactically and grammatically correct UML data definitions and/or instruction code. It is quite common,

however, for the module to be contextually incorrect since it is only after inclusion into a larger UML definition that it will make sense. For example, a module could contain the visual model definition given in Figure 4.4 which would be imported into the `Base` universe definition as shown in Figure 4.13. Note that the name of the element in the `visual.umm` file is not actually valid because it is not satisfying a previous stub declaration. Therefore, an attempt to parse this file on its own will result in an error. However, when it is imported into the definition contained in `base.uml` the result is perfectly valid.

```
// Filename: visual.umm

ELEMENT Visual
{
     // Element definitions...

     PROPERTY    surfaceList : Surface[];

     FUNCTION    Read( filename : STRING ) : BOOLEAN;
     FUNCTION    Write( filename : STRING ) : BOOLEAN;
}


// Filename: base.uml

UNIVERSE Base
{
     ELEMENT Models
     {
           IMPORT      "visual.umm"

           ELEMENT     Aural;

           PROPERTY    visual : Visual;

           // Etc...
     }

     // Rest of definition...
}
```

Figure 4.13 Importing a module.

The naming of files is left up to the discretion of the user. However, in this example the `.uml` extension is used to indicate a valid UML description, whilst `.umm` is used to indicate a module with potentially contextually invalid contents.

Code that is often re-used, in much the same way as traditional object-oriented classes, may be placed to best effect in modules. These modules may also be imported and instanced in the same way. A common use is the encapsulation of services, for example basic system calls. Rather than use two statements to import and instance the code, both may be done at once using, for example, `IMPORT "visual.umm" WITH visual`. This takes the top-level element in the file, in this case `Visual`, and declares a property with its type.

4.4.4 Summary

This section has presented an analysis of potential candidates for a modeling language. Due to some unique requirements the existing languages were deemed inadequate and the most important features of a new language, UML, were presented. UML is composed of a data definition language and an instruction code language. For a complete and formal description of the UML data definition grammar, please refer to Appendix A. An implementation of a UML interpreter is presented in chapter 5. The rest of this chapter describes the remainder of the integrated modeling/simulation system.

4.5 System Architecture

This section describes the structure of the proposed solution to distributing the universe simulation. A system overview is presented first, followed by a detailed description of the system's operation and concludes by separately addressing a couple of the key design issues.

4.5.1 Universal Simulation Node

The proposed building block for the Universal Simulation System, USS, is the *Universal Simulation Node* (USN). The USN has some important properties:

- It is capable of managing a complete simulation on its own without the aid of other USNs.
- Distribution falls within the near/tightly-coupled classification. This may range from a tightly-coupled multiprocessor system within a single chassis or a fast LAN connecting otherwise independent resources.
- The amount of bandwidth and computational power consumed by the simulation is at its highest at this level.
- Participants in the universe simulation use a USN as their gateway into the simulation.

Multiple USNs may be connected together to provide interoperability over near distances (Figure 4.14). This may be used to distribute an intensive simulation or to provide access for multiple participants to a single simulation (one or more participants would be present at each USN). The bandwidth used on the connections between USNs will be substantially less than at the USN level to reflect the (probable) change in network medium and nature of use. Such a grouping of USNs gives us a complete USS. In the remainder of this thesis whenever a node is discussed, it is actually referring to a USN and, similarly, a system corresponds to a single USS.

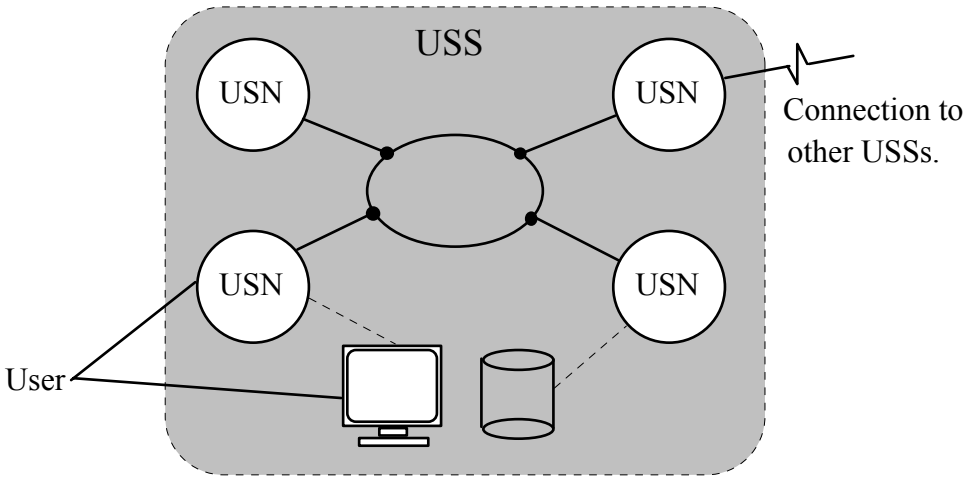


Figure 4.14 Example structure of a Universal Simulation System.

4.5.2 Universal Simulator System

A USS may be built from just one USN but this is generally inefficient due to the number of tasks that a fully configured USS must perform and the overheads incurred by each task. Distributing the workload between several nodes is more efficient. The tasks that a USS must perform are:

- Managing local input/output devices, e.g. joysticks, 3D mice, image generators.
- Handling communication with other USSs.
- Executing the simulation.

4.5.2.1 Essential Components

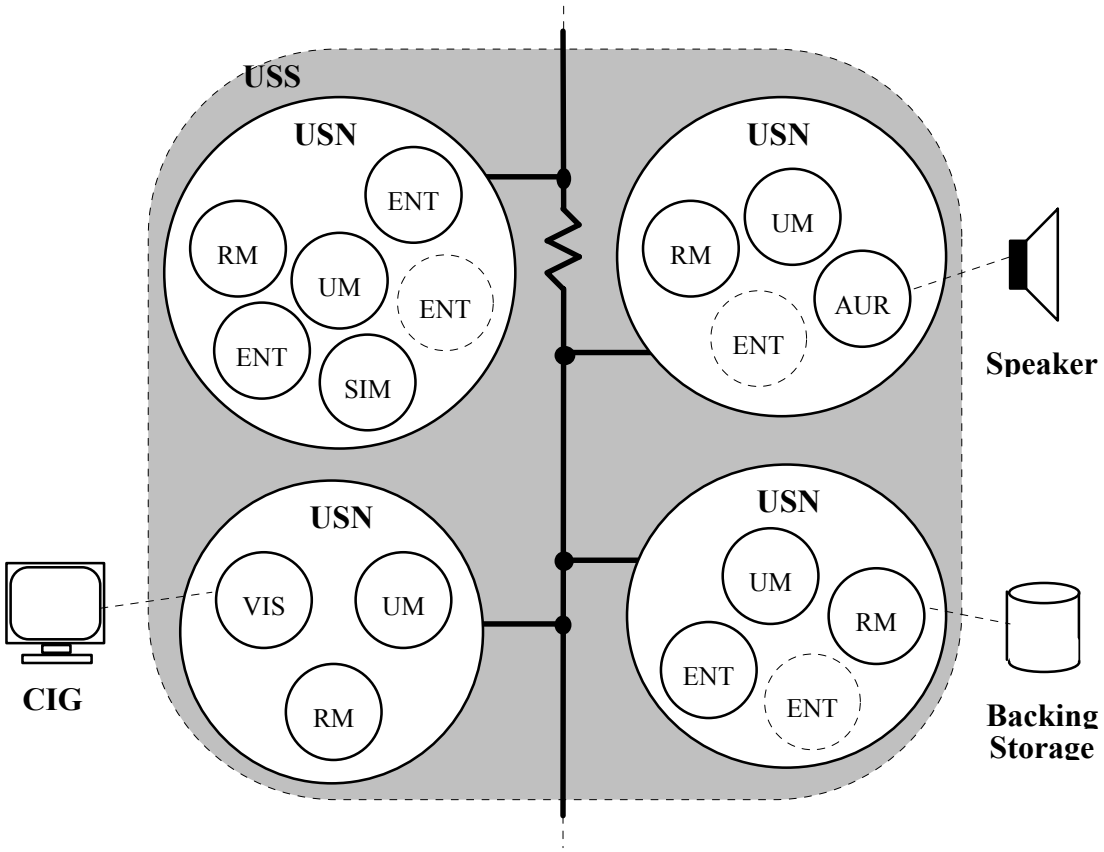
These tasks are undertaken by a number of different software components which all have a defined role. Each USN has a *Resource Manager* (RM) that is responsible for monitoring CPU usage, memory usage, controlling access to backing store and moderating the use of input/output devices to those processes that request them (Figure 4.15). At any time the RM is capable of providing information on the loading of the node and processing requests for other services. In essence, the RM contains the local scheduling functionality.

The *Universe Manager* (UM) is present in one form or another on every USN in the system. The UM of one node in each system is designated *master* and is responsible for communicating with the UMs residing on the other nodes in the system and also between other master UMs on other systems.

A universe consists of many autonomous entities (ENTs) which are implemented as separate processes. Each entity falls under the control of the node's UM (working in conjunction with the RM) which is responsible for scheduling the ENTs so that they are not starved of resources and can perform their work in time for the next simulation time step.

4.5.2.2 Optional Components

The three components UM, RM and ENT are the minimum required to form a USN and therefore support a simulation. Although entities may sample input devices, the simulation has no displays which makes this configuration of limited usefulness. Typically a visual and/or aural representation is given to entities within the simulation and there must be some way of making use of a CIG or sound equipment. This link between the output devices and the simulation comes in the form of special-purpose *Managers*. A manager monitors the information flow in the simulation and takes actions according to its purpose. The three managers described below are commonly used although others may be added without restriction.



Key

RM	Resource Manager	AUR	Aural Manager
UM	Universe Manager	VIS	Visual Manager
CDM	Spatial Integrity Manager	ENT	Entity

Figure 4.15 Example organisation of a USS complete with populated USNs.

If the system requires the use of a CIG then a manager has to be present in order to control access to it. One such manager is the *Visual Manager* (VIS) which runs on the node that the image generator is connected to. VIS provides services for representing and managing any part of the visual representation of the universe.

In the same way, the *Aural Manager* (AUR) is tied to a node with acoustic rendering equipment and provides services related to the aural representation of the universe.

The *Spatial Integrity Manager* (SIM) monitors the state of the universe being handled by the USS and notifies the relevant entities when there has been a breach of their spatial integrity, i.e. a collision. Response to these events are handled by the entities themselves.

Each UM can also support a *Console* which is essentially the hybrid of a manager and an entity. A console is forwarded the most important messages and provides a convenient way of collecting statistics. It may also be used to trigger certain events in the system.

4.5.3 System Organisation

There are no restrictions imposed by this architecture on how these components should be organised. Multiple specialised managers offering the same services can also be supported. The vast range of available processing power and communications bandwidth prevent the creation of a set of rules. However, it is possible to speak in general terms and provide according guidelines:

4.5.3.1 Near Tightly/Loosely-Coupled

A real-time distributed simulation's two enemies are the lack of bandwidth and communications latency. When transmissions between system components occur within the same physical machine then a given set of protocols may be used to communicate between certain processes. Assuming the configuration in Figure 4.15, this would mean that each USN could be attached to one processor, or maybe even a small farm of processors, communicating via a high-speed data bus. Passive partial data replication and complete computational distribution are used at this level.

As LANs increase in available bandwidth, it is possible to use these same protocols over a larger distance, latency permitting. In such a case each USN may reside on a different physical machine using, for example, fibre-optic cable as a transport medium.

4.5.3.2 Near/Far Loosely-coupled

There comes a point, however, when either the bandwidth is too small or the latency too great. As latency increases, use of the original protocols typically becomes less and less practical. To overcome this problem, networked USSs are connected and information to maintain synchronicity between these isolated systems is sent between them. An example of such information is that representing the interaction of participants at one system with other participants on another system and their influenced changes in the environment (section 4.5.4.10). In other words, total data and computational replication are used.

4.5.3.3 USS Networking

Those systems that use broadcast/multicast (section 2.4.4) have adopted a protocol that can compensate for the occasional missing packet. Data is sent regularly and is sufficiently

detailed that the lost information may be reconstituted or replaced by the succeeding messages. However, a lost message can result in temporary invalid behaviour which may have undesired side-effects.

By restricting the information that needs to be sent between systems to the bare minimum, i.e. level 2 behaviour distribution, the bandwidth required between systems is reduced proportionally. In an ideal world this information would be sent between systems using a low-overhead mechanism such as multicast. However, unless a reliable datagram protocol is available a lost message could have a profound effect. A message containing higher behavioural information is sent less frequently and failing to process it would effectively lead to that system running a different simulation to the others. Therefore, in both cases, there is a need for a *reliable* message delivery service.

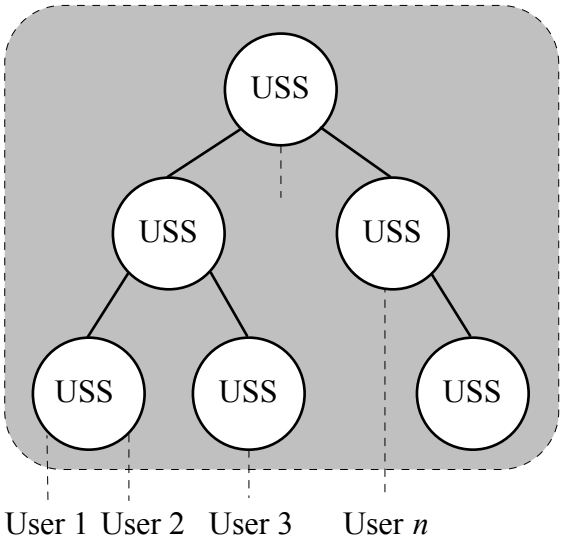


Figure 4.16 Hierarchical structuring of USSs.

Multicast is not widely available and the advent of reliable multicast services will take even longer to realise. Therefore it was decided to investigate a solution using point-to-point links with a view to future reliable multicast availability. If there are a large number of systems all participating in the same simulation, then the network of point-to-point connections between all of these systems would resemble a spider's web (section 2.4.1). To reduce communication overheads, a hierarchical network of systems may be constructed (Figure 4.16) such that any message to be sent outside a USS is sent to its parent and its children. The parent and children then determine if the message should progress further.

Since it is perceived that the information sent between systems is of relatively low bandwidth, the burden placed on each system for routing should be manageable. Unfortunately the latency this introduces may be insurmountable if the number of systems arranged in the hierarchy becomes too large. However, there is little alternative at this point in time. Interestingly, Bhagwat *et al.* (1994) have proposed a tree structure as the solution to scaling the error control mechanism used in reliable multicast for WAN usage. Certain nodes in the tree are assigned the responsibility of distributing the data reliably to the sub-trees rooted at these nodes. A tree structure is already used by the MBONE (Pullen, 1994), therefore there seems to be a need for a tree structure, regardless of the communication mechanism used, in order to cope with the transmission over long distances, reliable or not.

The amount of data generated by continuous live streams of audio and video would put a significant burden on any such organisation using point-to-point links and software routing processes. Fortunately, this is one type of information that can tolerate lost packets with few side-effects and therefore must be sent using conventional (unreliable) multicast techniques.

4.5.4 System Operation

Although each of the basic components (UM, RM, ENT) are separate processes, none can operate without the others and their functionality reflects the required interactions between them. Therefore, rather than fully describe each process in turn, a more function-oriented approach has been taken in this section concluding with some information on the common special managers. Implementation issues are discussed in the next chapter but a short note is provided in the following descriptions where there is an important decision to be made.

4.5.4.1 System Initialisation

The first USS started is at the root of the system hierarchy. The first process started within any system is the master UM (MUM) which then waits for its child, or slave UMs (SUMs), to connect to it using *activation* messages[3]. When all SUMs have connected to their MUM the system is ready to receive connections from its child systems in the system hierarchy (if any). Once these have been made it connects with its parent system unless, of course, it is the root system in which case the network of systems is deemed to be active. All inter-system communications are performed via the MUMs in each system.

Once a UM has connected with its parent, be it another UM or a USS, it starts its local RM and any other special managers configured to run on that node. Once the managers have established a link with their UM they provide it with a *service ID* which represents the type of manager they are and the nature of their services. The same service ID is shared by those managers providing an identical service (although their implementations may differ). Apart from the RM which has a service ID of 0, the UM does not know what ID matches which service, nor does it need to (section 4.5.4.5).

The next stage of the system initialisation is to parse the UML definition of the universe. A copy of this definition is sent to each specialised manager and forwarded to slave UMs. These managers then register interest in any parts of the definition that they wish to monitor with the UM (section 4.5.4.6). At the same time the MUM completes the initial process creation stage.

4.5.4.2 Universe Creation

At this stage, the only processes left to create are ENTs. The MUM processes each `ENTITY` definition in the UML description and starts an ENT process to represent that entity in the simulation. The location of the ENT is determined in conjunction with each node's RM as discussed in section 4.5.4.12. The entity creation phase concludes with the execution of their `Construct` function.

---

[3] In the following sections, description of the UM's role will represent either a MUM or SUM unless stated otherwise.

4.5.4.3 Universe Simulation

After the creation process has finished, the MUM is ready to start the simulation proper. The beginning of each simulation step is marked by the transmission of an *update notification* message to each ENT, manager and SUM (Figure 4.17). On receipt of this message, the SUMs forward the message to their local ENTs and special managers. Each entity executes its `Update` function, sends any modified state back to its local UM and waits for the next update message.

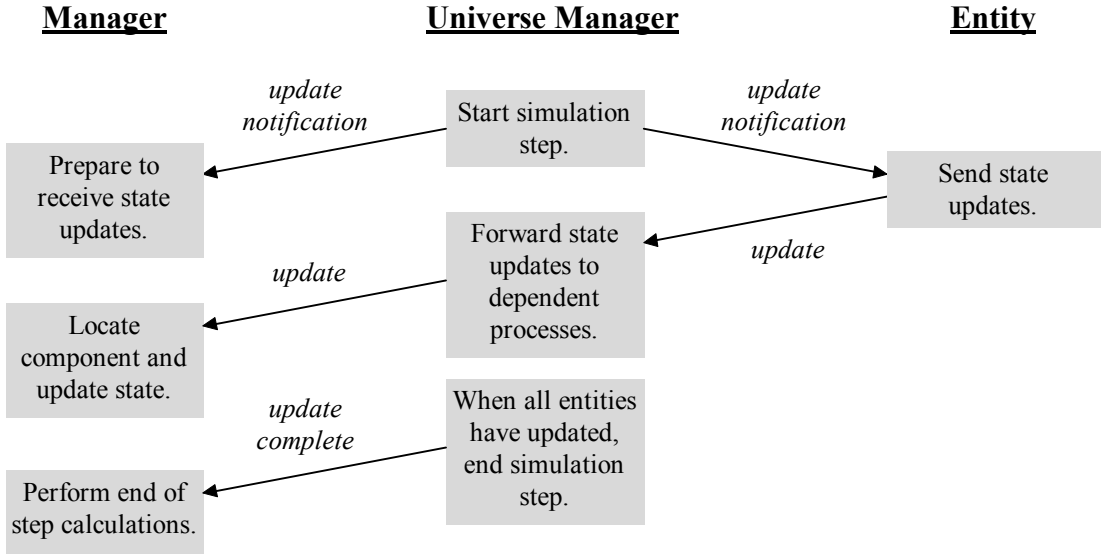


Figure 4.17 Order of events for a simulation update.

After receipt of an update notification message, each special manager waits for update messages to be forwarded to it via the UM. Once all messages have been forwarded, the end of the simulation step is marked by an *update complete* message which is sent to the managers only. When the managers have finished their work the update process begins again.

4.5.4.4 Master/Slave UM Relationship

Within a USS all information is completely distributed. This means that any event which occurs on one USN which may effect the system state must be reflected on the other nodes in that system. For example, if a manager on one node registers interest in a part of the UML definition with its local UM (section 4.5.4.6), that message must be communicated to the rest of the UMs on the other nodes. Messages sent by local processes that are intended for remote nodes are sent to the UM which acts as a router and forwards them to the MUM; from here they are sent to the correct node. Most messages are intended for all nodes rather than one-to-one communications and this mechanism provides a convenient way of implementing a pseudo-multicast facility (Figure 4.18).

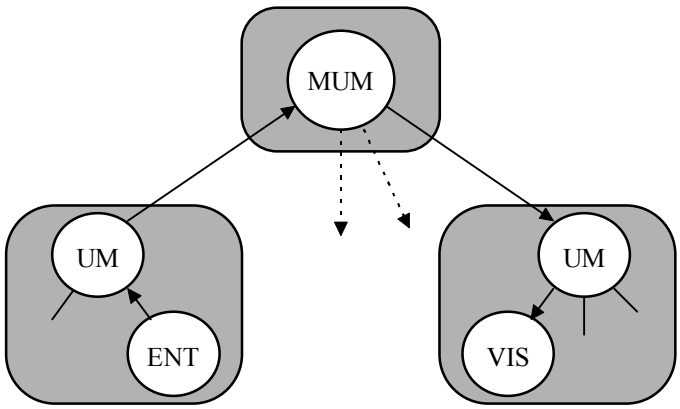


Figure 4.18 Possible communication path taken by a message sent from an entity to all managers.

Each UM (and RM) maintains a list of managers and entities on its node but the MUM also keeps a running total of the number of entities active on each slave node. Another difference between SUMs and the MUM is that the master node performs system-wide load balancing. In addition it manages the sole connection with the other systems. Live audio and video streams are dealt with separately in that the data packets containing this information coexist with simulation traffic but are only processed by the intended recipient (probably a special manager).

4.5.4.5 Locating Services

All processes throughout the systems, including the UMs, have a unique address called a *Universal Process IDentifier* (UPID). Examination of a UPID will describe the exact location of the process, its system, its node and its local address.

Any entity or manager may issue a *location request* which is sent to the UM in order to locate a particular process. The search may be restricted to the local node or permitted to extend throughout the system. If the search target is an entity then its name is given whereas a service ID is used for a manager. Should the service not exist locally and a system-wide search has been asked for, then the location request is forwarded to the MUM/SUMs. A successful search results in the return of the target process' UPID. The decision of which manager to use is left up to the entity to negotiate. Searches using a UPID as the key can also be performed and result in the return of either an entity name or a manager name and service ID. If multiple managers offering the same service are located, then all of their addresses are returned. Once a process' address is known, messages may be sent to it either directly, if it is on the same node, or indirectly using the UMs as routers.

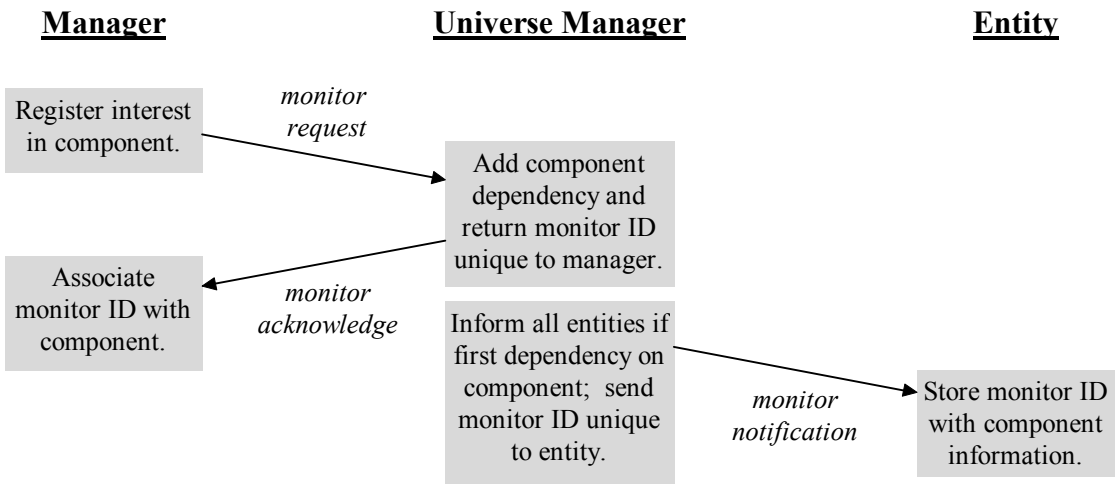


Figure 4.19 Procedure for registering interest in a UML component.

4.5.4.6 State Monitoring

The state of the simulation is represented entirely by the sum of all the individual entity states. The state of each entity is an instance of their local copy of the UML definition.

When a manager registers interest in a particular *component* of the UML definition it is said to be *monitoring* its change in state. After receiving the UML definition, a manager sends *monitor request* messages to its local UM which associates a *dependency* with the given UML component (Figure 4.19).

4.5.4.6.1 Unique State Identifiers

Somehow, the state sent by each entity to satisfy each dependency must be *uniquely* tagged such that each process throughout the *whole* system can identify it. The size of this ID can be approximated by the following equation:

```
ID size  = number of entities system-wide *
           number of entity's dependencies
```

Considering the potentially large number of entities system-wide and the number of dependencies that could be registered, this ID would have to be very big. How the ID is derived is also problematic. A centralised allocator could be used but this would not be very fault-tolerant. A network of mirrored allocators would be better but many IDs will be allocated and discarded throughout the lifetime of the system. The overhead incurred by interrogating such an allocator is too great for this to be a viable option. Basing the ID on the location of the entity is also impractical because entity's may migrate (section 4.5.4.12).

The chosen solution uses an ID which is unique between the UM and the process in question, whether it is an entity, a manager or another UM. As state updates are passed between processes, e.g. from an entity to interested managers, the UM inserts the correct ID for the communication. This may seem like an expensive process but, as shown in section 5.6.3.2, this may incorporated into the state distribution mechanism with negligible overhead.

This ID, known as a *monitor ID*, is returned by the UM in a *monitor acknowledge* message and is used in further transactions regarding this component. Multiple dependencies may exist for a given component, each one generated by a different manager - there is no point in a manager monitoring the same component more than once.

4.5.4.6.2 Synchronisation

If a given entity was created before monitoring of a particular component was registered (and it uses that component), then it will be sent a *monitor notification* message by its UM when that event occurs. It too will be given a *monitor ID* to be used in further communications. If a component has multiple monitors then only the first registration will generate a notification message. Conversely, if the entity is created after monitor requests have been processed, then the UM will *synchronise* it with the other entities by sending a stream of monitor notifications. Also, following entity migration the destination node's UM synchronises the entity to establish new monitor IDs.

4.5.4.6.3 Distributing Monitors

A copy of any monitor request received by a UM is forwarded to its MUM or SUMs and a similar process is undertaken to allocate a unique monitor ID between UMs. The remote UMs will then inform local entities as necessary. In this way, an entity on one node will know to send state updates for a component that is being monitored on a remote node. However, it does not know where the manager is, only that a manager is interested in its state.

4.5.4.6.4 Construct, Update and Destruct

At the end of the entity creation sequence, after all relevant data has been instanced, the entity's `Construct` function is executed which, when completed, results in one or more *construct* messages sent to the UM (Figure 4.20). Each message corresponds to a monitored component and holds the current state of that instanced component. Upon receipt, the UM looks for any dependencies on this component and forwards the entire message to the interested managers (with one proviso detailed below). At the end of the entity's update phase, similar *update* messages are sent: upon entity termination, a single *destruct* message is sent. Note that update messages are only transmitted if the entity has modified that part of its state since the last update notification was received.

When a manager receives a construct message it instances the monitored component and copies the state contained within the message. As update messages are received it updates its local copy of the state and deletes the instance if it should receive a destruct message. Upon receipt of these message types a manager executes its own construct/update/destruct functions. These functions perform some action which is unique to each manager, e.g. the update function may wait for an entity position change so its visual representation may be moved. At any time a manager may also get the current state of an entity's component by sending a *state request* to the UM which is forwarded to the entity. The state is returned in a message with the same structure as a normal update message.

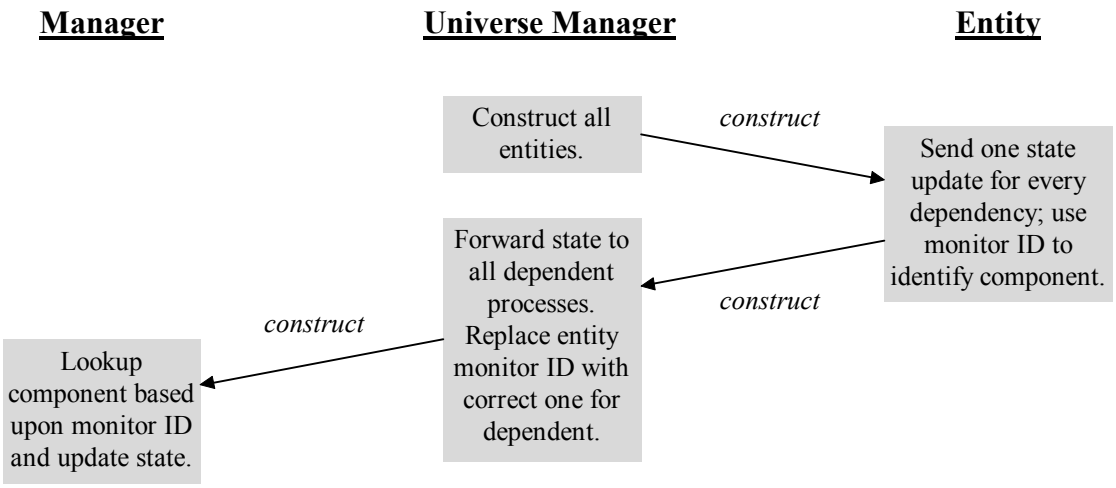


Figure 4.20 Sequence of events during entity construction.

As inferred in section 4.5.4.4, the local UM will send a copy of the relevant entity construct, update and destruct messages. Obviously, if those nodes do not have any managers running on them, then there is no need to send these messages at all.

4.5.4.6.5 Constraint Functions

A manager can also supply a *constraint function* (written in UML) to be associated with each component it is monitoring. Every time the UM receives a state update for a monitored component it executes the constraint functions (if they are present). If the dependency is with a local manager and the constraints are met then the state is forwarded to the manager, otherwise no message is sent. All dependencies with remote managers are represented locally as dependencies with other UMs and the evaluate-send sequence has two additional conditions. Firstly, if multiple dependencies for a single component exist with remote managers on the same node, then each function is executed in turn until one succeeds or all have failed. Secondly, if one or more of the dependencies for a given node do not have a constraint function attached, then the state is sent immediately without executing any of the functions. Constraint functions may be updated or added at any time by the manager that owns the dependency.

4.5.4.7 Localisation

WAVES filters messages upon reception so that only those entities in a viewable area associated with a given host are sent to that host. NPSNET splits the environment into a mesh of two dimensional hexagonal cells and uses multicast groups to ensure that only the entities within the local and neighbouring cells are processed. AVIARY's EDB provides a comprehensive range of services including collision detection and entity operations based upon volumes of space. One such volumetric service is the monitoring of a specified region of space for a client. When an entity enters, leaves, moves or changes whilst in that volume the client is notified and may take according action.

A criticism that Snowdon (1995) makes of the approach taken by WAVES is that a lot of bandwidth may be consumed for no real reason since all messages are only filtered at the destination. This is a valid point which USS does not suffer from. By using constraint functions, filtering is done at source which, combined with state updates that are only sent when a change has occurred, reduces the required bandwidth to an absolute minimum.

The localisation techniques used by both NPSNET and AVIARY "filter" based solely on position and volume. NPSNET does this merely to reduce the amount of entities it needs to process whilst the EDB also performs collision detection. However, in a USS a constraint function can be imposed on any component, not just one representing position. As part of the basic services offered by a USS, the UM has no understanding of what the UML description means, just that it is composed of constants, elements, properties, etc. Only the manager that specifies the constraint function needs to understand what it means. By abstracting the filtering process in this way, it is just as easy to receive information about entities within a given volume as it is to restrict messages to changes in an entity's colour. If only position changes are wanted for red entities, for example, then it is necessary to encapsulate the position and colour properties in another such that the constraint function may compare them.

Consider the common case of an entity moving through space, entering a volume monitored by a manager and passing through until it leaves at the other side. When the manager starts receiving messages because the entity has entered the volume, it needs to know the current state of all the components it is monitoring, not just the one that has just changed, i.e. their position. Similarly, when the constraint fails it needs to be informed so that the entity can be dropped from its calculations. To resolve this problem a constraint function has the *optional* functionality to issue a state request to the entity on behalf of the manager. When entering the volume one or more pseudo-construct messages are sent (one for each monitored component) and a pseudo-destruct message when leaving. Although the entity is not *actually* constructing and destructing it is as far as the manager is concerned.

4.5.4.8 Modifying the Universe Definition

The strongest advantage of using an interpreted language for modeling was that it facilitated modifying the definition of the VE. This may involve an addition to a given component, the deletion of part of its structure (or the whole component), or the definition of a new component (or part thereof). Whenever a change happens, regardless of its nature, every process in the system must be informed (with the exception of the RM). In addition, since this is a fundamental change in the simulation, it must be communicated to other systems simulating the same universe. Such a change is introduced by an entity (probably initiated by a user) and sent to the MUM in a *uml* message. The change is first parsed by the MUM and if this is successful a lock for the portions of the definition being modified is negotiated with the other systems. The new definition is then forwarded to the MUM's local entities, special managers, slave UMs and to any systems it is in contact with. All modifications are made system-wide within one simulation step. When the other systems have acknowledged that their modifications are complete, the lock is released. To accommodate for lags in the system and between systems, changes may be queued and effected at a predetermined time. This allows the changes to be transmitted to the furthest node/system and after all nodes/systems have the modification request in their position, the change is effected simultaneously.

If a component is extended then default values must be given to the newly added subsection. If part of an existing component is removed then accesses to this old information must also be removed. Addition of a new component outside the scope of usage by any entity, or not within the components being monitored, does not have any side-effects. These issues are dealt with further in section 5.5 which discusses the implementation of a UML interpreter.

4.5.4.9 Multiple Universes

The purpose of a UM is to manage the execution of simulations of universes described using UML. Entities form logical groups reflecting the universes they belong to although they may still execute on the same node. In order to support multiple universes, it is necessary to tag every message sent with a unique *universe ID* that must be processed every time a message is received by a UM or RM. The UM vets messages for entities so that they are never sent a message originating from another universe. It would be possible for managers to handle information from multiple universes simultaneously, but this might either be impractical, e.g. in the case of VIS, or inefficient, e.g. the SIM is a computationally expensive process. On the positive side, having the relevant information for all universes in one place simplifies the process of entity migration (section 4.5.4.12). Therefore, the designer of a special manager must make the decision to have one manager for all universes, or one manager per universe.

When an entity moves from one universe to another a destruct message is issued. The parts of its definition that it has in common with the destination universe are preserved (and their associated state) whereas the others are destructed as per usual. The entity then constructs in the target universe, building upon the partial state it has retained from the source universe by instancing those properties that are new to the entity and assigning default values. Finally the entity is moved from one universe group to the other. Note that there is no need to terminate and recreate the ENT process, just alter its state.

4.5.4.10 Multiple Users

Users are represented by entities that read input devices and take actions accordingly. Multiple users can be supported within the same system without adding any extra functionality. This is not true when users on different systems wish to interact. Each USS is totally replicating the computation and data yet each system has what are, in effect, wildcards - users. A user on one system must be represented on the others and their actions reflected, i.e. their behaviour must be modeled in some way. This goal represents a level 3 distribution which, as discussed in section 2.2.4.5, is not feasible since the decision making process is too complicated.

Consider the example of a user driving a virtual car. Sending changes in the car's position (level 0) over low bandwidth communications links is wasteful but highly accurate. Level 1 distribution can be achieved by approximating the dynamics of the car, i.e. a dead-reckoning model. This is not very accurate and can result in sudden changes in the modeled variables as updated parameters are sent by the real entity. Parameterising a user's actions over time, such as turning the steering wheel or pressing the pedals is also feasible. By triggering pre-programmed control movements it is possible to achieve level 2 distribution and an approximate representation.

However, representing a user that is walking around the environment, moving their arms and legs is not as simple. Limb positions could be approximated based on velocity but subtle movements would be lost. Given a set of animated behaviours such as “move forward”, “turn left”, “pick object up”, etc., then level 2 distribution could be achieved. But this solution shares the same problem with the previous example: how do you map the constantly changing data from the input devices into a series of pre-programmed movements?

It appears, therefore, that the level of behaviour modeling required depends on the method of interaction and representation utilised by each user. Since this is an area of research that requires a great deal of further work, USS does not impose one particular method.

All messages sent by an entity representing a user are tagged accordingly. They are processed in exactly the same way, except that when they reach the MUM they are also forwarded to any systems that are simulating the same universe. When an entity construct is received by the MUM on another system, a new *shadow* entity is constructed and its state taken from the message. This process functions in the simulation in the normal way until a destruct message is encountered. These are the only two messages that are always sent, any other type of message to be sent to the entity’s shadow must be specifically indicated.

By flagging update messages, all component updates made by the entity are forwarded to the shadow - use of this option is not recommended. Preferably, when modeling the entity a number of UML functions can be written that, when executed, will perform an automated manipulation of the entity’s properties. This could result in a position change or the triggering of a sound, etc. By redefining the shadow’s update function to exclusively call this function, animated behaviour is possible. Level 2 distribution may be accomplished by leaving the update function empty and remotely invoking these functions in a certain sequence to effect the desired result. These last two methods merely use the *uml* message to send UML code to the shadow entities and are issued within the real entity’s `Update` function.

4.5.4.11 Entity Lifetimes

Most entities are created when the initial universe creation occurs but they may also be created at run-time. An entity can only be created by a UM but creation requests can be made by other entities.

Entities may terminate (abnormally or naturally) at run-time and new entities not originally specified in the universe definition may be created. Notification of entity terminations are sent to all managers that were monitoring its state in *process notification* messages. Entity creation follows the usual procedure and requires monitor dependency synchronisation.

An entity may opt to save its current state to backing store before termination so that it may be loaded again when it re-enters the simulation. This mechanism is often used by users since they are not always present in a simulation.

4.5.4.12 Scheduling

When a process (including the UM) is created, it is allocated a *Resource Profile* (RP) which holds information about which resources it needs, how much and (if possible) when. A new process is given a default allocation of resources (or a hand-written specification) which is

modified and tuned during the execution of the simulation. At the beginning of each simulation step, all the entities and managers within the simulation are given access to the resources through a dynamic deadline scheduler so that they may complete their calculations for the current step.

Upon completion of each entity's calculations, information regarding the amount of resources that they consumed is processed by the RM, so it may adjust their scheduling parameters, if necessary. When a schedule entry is inserted, deleted or changed, some or all of the other entries must also be reallocated. Resource contention is accounted for in the scheduling. It is possible that a time will occur when completing all the calculations necessary within one time step is impossible. At this point there are four choices:

1. Flag that a fatal error has occurred and terminate the simulation.
2. Degrade the number or accuracy of calculations currently performed so that the final deadline may be met.
3. Degrade the simulation by extending the duration of the simulation period thus resulting in a lower simulation update rate.
4. Migrate the offending process to another node.

The first option is obviously highly undesirable, the second is fine in theory but implementing an entity with alternative computation paths based on complexity is more complex in itself and will require more memory to store them. Whilst an attractive approach for a manager, e.g. varying the accuracy of collision detection based on the time available, this could lead to different outcomes on different systems and hence different simulations. However, a slight variation on this technique would be to reduce the rate at which each process was updated. If, for example, an entity represented a very slow moving entity then updating it at 30 Hz may be excessive if no noticeable difference is made at 5 Hz (Wloka, 1993). Such functionality can be programmed into the entity without complicating the task of the UM further, i.e. an update is not returned until a pre-defined threshold is reached. Extending the duration of the simulation step is a valid option but should only be used if the fourth and final option is not possible.

By periodically interrogating each RM, the MUM can determine whether the workload on any node is too high and that an ENT should be moved to another node. (The RM includes itself in the list of resource consumers when calculating the total utilisation for each resource.) The actual entity is chosen by the RM and its current RP is sent to the MUM so that it may determine which node has the best chance of accommodating it. If the chosen node cannot schedule the entity, e.g. due to resource constraints, it rejects the *migration order* and the MUM chooses another node. Alternatively, if the RM determines that a particular ENT will exceed the available resources before the next load check, it may send a *migration request* to the MUM containing the entity's RP. Stankovic *et al.* use an algorithm whereby each node is responsible for finding a new home for a process (section 4.3.1.4); USS takes advantage of the fact that all inter-node communications are routed through the MUM. It is only a small step from this position to delegating all responsibility for locating a new node to the MUM.

Once a decision has been made to migrate an entity, an ENT process is created on the destination node. The entity's complete state is then packaged up (in the same way as smaller sections are for construct and update messages), sent to the newly created destination ENT

process in a *migration state* message and followed by its current RP. After this transfer has been completed the old process is terminated. The migration is scheduled to take place after the entity has completed its update so that the current simulation step is not affected. All managers that are dependent on any part of the entity's state are sent *migration notification* messages to inform them of the change. Any other messages that should slip through this net, e.g. direct communications with another entity, are forwarded by the UM and the sender is notified of the move. The MUM does not keep track where each entity is, only how many run on each node. The only time location information is needed is during an entity migration, at which point an entry is added to the *migration list*. This entry details the entity name, source node, target node and original address. Once a migration has completed the entity's entry is removed from the list.

Managers may, of course, also consume more and more resources but it is neither feasible nor efficient to migrate them to another node. Firstly, they possess a large amount of state information, albeit copies, and they may also be tied to specific hardware in that node. Finally, the time it takes to move a manager will by far exceed the time taken to move a single entity and may, in itself, cause problems with scheduling.

The RPs for UMs reflect that they are more demanding than most of the other processes in the system. In fact, the node holding the MUM will most likely have fewer managers and entities due to its increased administration responsibilities.

4.5.4.13 Resource Profiles

Process-specific resource consumption and scheduling requirements are held in an RP. In fact, a process maintains a *resource history* which stores a copy of the previous RP, the current RP and a prediction of future resource requirements. The RP also shows whether exclusive access is required to a resource or whether shared access is permissible.

An RP is composed of the four basic resources that each node can possess: *computation*, *memory*, *backing storage* and *network*. The capacity of a given resource is measured in a different way each time. In the same way, determining the utilisation is specific to the resource and is represented by a percentage. In all cases, the limitations of the node's physical architecture, such as internal bus speed, are incorporated into all of the ratings given.

Computation is gauged by both an integer and a floating-point rating. The CPU type and statistics are also held, possible CPU types are: Reduced Instruction Set Computer (RISC), Complex Instruction Set Computer (CISC), Vector, (specifying size of Data and Instruction Caches) and CPUs with specialised extensions. The presence of a Floating Point Unit (FPU) is explicitly indicated since floating-point operations may be emulated in software. If a FPU is not present then each rating represents the CPUs performance given only an integer or a floating-point workload. The CPU statistics are, in general, only used for scheduling purposes.

Memory is rated by its size in Megabytes (Mb) and its access time in nanoseconds (ns). The total amount of memory being used is periodically recorded.

Backing storage is also rated by how large it is, its average access time in microseconds (μs) and cache size - these last two statistics are combined to provide a convenient rating. A record is also made of how much disk space is being used. This is the only optional resource.

Network capacity is measured in Megabits per second (Mbps) after taking into consideration the protocol overheads. Calculating its utilisation is somewhat tricky without operating system support but can be approximated based on the number of messages sent and their average size. In order to support live audio and video feeds, it is necessary to include the bandwidth consumed by these transmissions when calculating the node's total network utilisation. This information is collated by the UM (through which all messages pass) and periodically communicated to the RM.

Each resource is run at a percentage of its maximum to establish a threshold beyond which some load balancing action must be taken. The threshold for each resource is set independently, each specified as a percentage of the resource's maximum potential. The CPU has two thresholds, one each for integer and floating-point capacity. To prevent a situation whereby the slightest change in resource consumption results in a migration request being sent to the MUM, a latency factor is associated with each resource. If the resource should remain over-utilised for longer than the specified time, or there is a continuous dramatic increase, then action is taken.

4.5.4.14 Input/Output Devices

A node may have one or more peripherals attached to it, such as mice, joysticks, 6 d.o.f. tracking systems, a sound system, a CIG and so on. All of these are classified as resources and access to them is monitored by the RM. The capacity rating of each of these resources can differ by so much and can be measured in so many different ways that the RM does not attempt to hold this information. Only a percentage utilisation is stored which is provided by each of the specific drivers for the given resources (as is their initial rating). Such ratings are resource dependent and cannot be compared between different resources.

The device drivers would be best organised as tasks within the RM itself but this can cause the RM to become a bottleneck within the system, therefore *Device Drivers* (DDs) have an identity of their own with the system (what form this takes depends on the implementation). All access to the devices is through these DDs who keep the RM informed on their utilisation. Common roles for DDs are providing access to serial and parallel ports, disk controllers, digital I/O, Analogue to Digital Converters (ADCs), etc. Often DDs are provided with higher functionality such as a filing system, mouse drivers, joysticks, 6 d.o.f. trackers and so on.

Implementation of the DDs may take the form of a separate process when the resource can support servicing multiple requests simultaneously, but more commonly it may be provided as a library which may be incorporated into a software component with a high level of functionality. For example, VIS requires access to the CIG and having a separate process in between it and the CIG hardware will only cause a performance loss. It is far more efficient to incorporate the DD into VIS and inform the RM that this resource is no longer available since VIS has exclusive access

4.5.4.15 Visual Manager

The Visual Manager, VIS, provides a standardised interface to all CIGs. These may be special hardware attached to the node either as an integral part of the node's hardware or an extension to it. Alternatively, VIS may incorporate a 3D software library which is capable of interactive performance such as RealityLab™ (Microsoft RenderMorphics Ltd., UK), Renderware™ (Criterion Software Ltd., UK) or BRender™ (Argonaut, Inc., USA). The actual underlying technology used for image generation and their specific interfaces are hidden from the rest of the system. In both cases, VIS requests exclusive access to the dedicated hardware or video card via the RM. One of the many services that VIS offers is the ability to use more than one CIG channel, i.e. it can drive multiple instances of a given CIG and even many different makes of CIG simultaneously.

The introduction of a special manager such as VIS necessitates the inclusion of a standardised UML definition to represent the information it needs. For example, the property instance `models.visual` (Figure 4.3) of the `Visual` element (Figure 4.4) and the `position` property. After creation, VIS would notify the UM of its wish to monitor all instances of these properties. As each entity was constructed, so VIS would take the visual representation and construct a new visual object. In most cases an entity would rarely modify its visual representation, so the following updates would usually only consist of position changes.

Each VIS manager associates a viewpoint with one or more CIG channels. For example, a non-frame sequential stereoscopic display would use two channels, one for each eye. One viewpoint would be used, modified slightly to produce the correct projections for each eye. It is likely that there will be more than one VIS manager in a given system, e.g. one per user, possibly more than one per node. When an entity requests the location of a VIS manager from the UM it receives a list of the active VIS managers. A manager may be assigned to an entity after which it will not be available for use by other entities until it is released. Once service access has been restricted to one entity, that entity may manipulate the viewpoint's parameters, such as position, orientation, aspect ratio, etc.

To prevent itself from receiving information about every entity in the simulation, VIS associates a constraint function with the `position` property that specifies a volume around the current viewpoint. As the viewpoint changes, the manager updates the constraint function associated with the component dependency held by the UM. In order that the network is not flooded with constraint function updates, they are only sent when the distance of the viewpoint from the centre of the current volume reaches a certain threshold. Upon entering the volume, an entity sends (pseudo-) construct messages which hold the entity's current position and its visual representation. When leaving the volume a single (pseudo-) destruct message is sent indicating that the entity should no longer be considered for rendering.

To avoid the transmission of visual representations as each entity constructs, it would be desirable to provide a library of models, one of which would be referenced in the `Visual` element, thus superseding the detailed geometric description. The library would be accessible by all VIS managers and common models could even be cached to reduce library access. This technique makes it difficult for an entity to modify its representation at the vertex/polygon level and therefore should be provided as an option to the current method and not a replacement.

4.5.4.16 Aural Manager

AUR gains exclusive access to the pertinent hardware for generating sounds via the RM. Copies of all information regarding the aural representation of the universe is held within AUR and changes to it are monitored by the manager. In the same way that VIS provides a generic graphics interface, AUR provides a generic interface to generating sounds and thus may support many different hardware and software solutions. Changes in information that affect how the sound is generated, e.g. movement of an entity, are sent automatically to AUR using the usual methods. Constraint functions are used in the same way as VIS to restrict the number of entities that must be processed. Typically the volume monitored by AUR will be different to that used by VIS. For example, when sitting in a closed room with no windows one cannot see outside that room but it is probable that one will hear sounds originating outside.

4.5.4.17 Spatial Integrity Manager

For reasons of speed and efficiency, one of the most computationally expensive processes is implemented as an optional manager. No particular method of intersection testing is advocated in USS since the methods available consume varying amounts of resources (Webb and Gigante, 1992; Bouma and Vanecek Jr., 1991; Cameron, 1990). It is important, however, that the same method is used on all systems.

At the minimum, details of the volume that an entity occupies are necessary along with position and orientation information, whilst a more ambitious SIM might require a velocity vector. For more accurate determination of collisions a detailed geometrical description of the entities involved in the collision would also be needed so that the exact point of collision may be pinpointed (Zyda *et al.*, 1993). Utilisation of behavioural information for each entity is another possible approach and can be shown to reduce network traffic since only behaviours need be transmitted rather than continuous positional information. Obviously the more accurate collision detection used, the more time and space the process requires. By providing a generic interface, the type of collision detection method may be changed depending on the resources available, taking advantage of more powerful hardware. Once a collision has been detected, each colliding entity is sent an *entity interaction* message which holds the UPIDs of the other involved entities. The entity that caused the incident is nominated to co-ordinate the resolution process.

The load placed upon the SIM may be relieved by using multiple co-ordinating managers in a manner similar to AVIARY's EDB. When the volume is split the original SIM modifies its existing constraint function whilst the new SIM lodges more monitor requests complete with its own constraint functions.

4.5.4.18 Console

Commands may be entered through a simple command-line interpreter. These are mainly interrogative but a console may force the destruction or creation of entities at run-time. Other manipulative operations include the purging of references to a given process from the UMs internal data structures (and those on other nodes) which is useful when a process has abnormally terminated. However, the console does not actually take part in the simulation.

4.5.4.19 System/Node Lifetimes

Nodes are users' gateways into the simulation and it is probable that they will not be powered on all of the time. Therefore a mechanism by which nodes may enter and leave the simulation is required. When leaving the system, all entities related specifically to users on that node are terminated. Once this is complete the remaining entities are migrated to other nodes and any special managers inform their clients that they are terminating. Finally, when the only processes that remain are the RM and UM, a *deactivation* message is sent to the MUM and the node ceases activity.

Re-entering the system is achieved by proceeding through the usual initialisation steps (section 4.5.4.1). The MUM may then utilise the node's resources for scheduling purposes, resulting in entity migrations.

When a system leaves, it sends the master USS a deactivation message which is the cue to remove all processes representing users on the parting system from the simulation. Joining a running simulation means that the current UML definition must be obtained from another USS, complete with current states for all entities. For this reason, joining a established network of systems is only practical for very small simulations and, even then, not recommended.

4.5.5 Time Management

As conjectured in section 2.4.5, an explicit time progression model is used within a USS and an implicit time model is used to synchronise multiple systems.

4.5.5.1 Explicit

The explicit model takes the form of the update notification/complete message pair which are scheduled to occur at the same point in each simulation step. This is not to say that the dependency on the system clock has been removed from the system. In fact the opposite is true since all the scheduling is performed and monitored in relation to clock time. However, there is no need to synchronise the clocks between nodes since the execution of the schedule for a node is done locally. Each simulation step happens in a relatively small amount of time, especially for real-time simulations. Therefore, at this level oscillator drift will not effect the timing of the schedule.

There is no requirement that time is modeled in the VE but the usefulness of an environment that does not use time in some way is dubious. The relationship between simulation time and real clock time can be modeled in a UML function, e.g. `time()` in Figure 4.3. This function would use an expression based upon the current real clock time and the current step count. Using a function means that this relationship may be redefined at run-time by providing a new function definition. This change would, of course, be sent to all other processes in the system.

4.5.5.2 Implicit

Synchronisation of time between systems is more problematic. Each system uses total replication of computation and data, so it may seem that tight synchrony is not all that

important. There is one important exception which is that the users are not replicated. One user's actions in one system must be reflected in the other systems through their shadow and *vice versa*. In order to prevent lag from destroying effective interaction between these users, the systems must be synchronised to the same simulation update. Only then is there a chance that behavioural information sent from one system to another can be incorporated into the current update.

There currently seems to be no good solution to this problem. SPS is perfect for the task, providing the ability to synchronise with 167 ns, but at the time of writing one receiver can cost upward of US$500 (Dana, 1995). NTP is commonly used between systems using TCP/IP although this is not a requirement, but access to a machine that keeps accurate time is. In fact, a number of the world-wide primary NTP reference sources use radio or wire to synchronise with national standard time. Ensuring all systems world-wide have the same time would currently necessitate access to the Internet. More importantly, the greater the synchronisation accuracy, the longer the period required to achieve it (a few hours) and the increased bandwidth.

4.5.6 Fault Tolerance

Problems can occur at different points in a system and in different components. The policies used to handle these events are presented below.

4.5.6.1 Software Component Failure

If a manager has failed then it may be restarted on the same node and its state copies gradually reconstituted from the following update messages. If this is not sufficient then a state request can be made to the UM for detailed state information from each entity.

A restarted entity cannot be revived in the same way. Either it must start with its original state or obtain the current state from one of its clones in another system.

4.5.6.2 Hardware Component Failure

Individual hardware component failure may be tolerated by migration of the dependent process to another node in the system. If the failed component's functionality is not duplicated anywhere in the system, then either the process must attempt to continue execution without it or be terminated.

Should the replacement of the faulty hardware require the whole node to be shut down, then all processes must be redistributed to other nodes.

4.5.6.3 Node Failure

Loss of communications with a node requires the simulation to be frozen immediately. There are then two options to choose between. Firstly, simply wait until the node has been recovered and then continue the simulation. Secondly, the MUM re-creates those processes that are on the failed node elsewhere in the system. The current state of these entities can then

be acquired from another system running the same simulation. Once state has been restored the simulation may continue again. When the faulty node is restored its entities are removed and the system load re-distributed.

4.5.6.4 System Failure

Failure of a communication path with a system will not affect the other systems. If only external communications have failed, then the simulation on the isolated system is frozen to prevent the users from making any changes to the environment that would have to be abandoned. When the link has been re-established the system synchronises and enables the simulation again. This synchronisation process can be quite lengthy: entity deaths and births must be checked, entity states updated from clones, user interactions on other systems reflected locally, etc. In order not to overload any one system it would be possible to obtain this information from a number of systems throughout the network.

4.5.6.5 Summary

Application of those recovery techniques that require collaboration within another system is problematic. Bandwidth between systems will be at a premium and the latency greater than node-to-node communications. Therefore these procedures will undoubtedly be prolonged affairs but, unfortunately, there is little alternative. Even those policies for recovering a single node or software process may take longer than a simulation step. Thus the local simulation will suffer until recovery is completed.

4.5.7 Access Control

There is no access protocol built into the basic components of a USS. However, there are a number of system features that provide some methods of restricting the options.

4.5.7.1 Resources

At the most basic level, all accesses to system resources are granted by the RM and it is not possible for a process to bypass this mechanism if it wishes to be scheduled for run-time. Access to specific devices can be pre-allocated to managers and restricted by location. For example, a VIS manager is given dedicated access to a CIG and is required to run on the same node.

4.5.7.2 Location

Each message includes the UPID of the sender and therefore service requests can be rejected based on location, e.g. a VIS manager may only want to deal with requests from entities on its own node. If an entity or a manager should be concerned about the sender's identity then its full identity may be discovered by issuing a location request.

4.5.7.3 Snooping

The worst security risk is that an unwanted process will examine an entity's state by monitoring the state updates. The only process that can do this is a manager and, unlike entities, these cannot be started at run-time. Therefore, to introduce a bogus manager into the system would require the alteration of configuration files and a system reboot. Neither of which would likely go ahead unnoticed.

4.5.7.4 Insulation

Since UML code may be introduced at run-time there is a potential for misuse, however, there is very limited access to the system services. A typical entity will only require access to the system clock, location requests, sending and receiving UML code. UML therefore acts as an insulating layer between deliberate or accidental intent and the low-level operation of a USS.

4.5.8 Feature Summary

A summary of the system architecture's key aspects is given below.

4.5.8.1 Structure

A USS is made up of a network of USNs. The decision of whether to have a group of nodes forming one system or a network of one node systems is based upon the computational power of each node, the bandwidth of the network and the distance between nodes, i.e. the length of the propagation delay. Low computational power and high bandwidth lends itself towards a network of nodes whilst high computational power and low bandwidth is better suited by a network of systems. Since there is no reliable multicast transport mechanism readily available, point-to-point communications are used to ensure 100% reliability.

Passive partial data replication and complete computational distribution are used within a system. A network of USSs use total data and computation replication.

4.5.8.2 Services

The UM and the RM provide the core services whilst ENTs are used to execute a universe simulation written in the modeling language UML. Special managers such as VIS and SIM are not *needed* to run a simulation but are often used since they can encapsulate useful services, e.g. image generation and collision detection. The UM understands how the information in a UML definition is structured but does not understand what it means. Only ENT processes and special managers know what the data means and what to do with it.

The UM is at the heart of the architecture, either in the shape of the MUM or a SUM. The key services that a UM provides are:

- Message routing between local processes and remote nodes.
- Process/service location and identification.
- Processing of monitor requests placed by managers and adhered to by entities.

- Managing the introduction of changes/additions to the VE description.
- Managing migration of a local entity.

Additional functionality unique to the MUM:

- Managing node activation and deactivation.
- Controlling initial simulation creation.
- System-wide scheduling including the coordination of entity migrations.
- Managing general communications with remote systems.
- Forwarding of local user information to their shadows on remote systems.

The RM works closely with the UM to provide an execution environment for the simulation. Services include:

- Controlling access to the node's resources.
- Scheduling of all processes on a node such that they complete execution before the end of each simulation step.
- Advising the local UM and the MUM on the node's loading.

4.5.8.3 State Management

The instance data of a universe is the sum of all the states owned by each entity. The owner is the only process that is allowed to modify the state. Managers cannot modify any state information directly, they can only examine it. A manager may, however, indirectly cause a change in the entity's state through execution of one of the entity's UML functions. This job demarcation removes the need for any locking mechanisms.

Managers register an interest in a particular component of the universe description. Any changes made by an entity to their instance of that component are relayed to the managers via the UMs. The information in the universe may be further filtered by specifying a constraint function which is applied to each update sent by the entity. If the constraints are met then the message is sent to the manager.

4.6 Summary

This chapter has presented the requirements of a system capable of distributing and simulating a VE, its design restrictions, real-time issues and the implications of these features. The proposed design begins with the presentation of the language used to model the VE which is based upon an interpreter to provide the utmost flexibility. The presented system design exists to execute the simulation described by the modeling language whilst transparently distributing it over a network of machines (nodes). Nodes are grouped into systems based on their ability to support complete computation and passive partial data distribution. Clusters of these systems are consequently interconnected by lower bandwidth links and only information unique to any given system is communicated to the others. A number of required software components run on each node to provide administrative functions and an execution framework. Each entity within the simulation is embodied in a process that represents part of the universe's state. Managers provide specialised services to entities within the system by monitoring changes in portions of the entity's state. All work is scheduled using a local

scheduling policy and a system-wide policy, ensuring that the load across all nodes stays balanced through the use of process migration.

Now that both the modeling and simulation execution aspects of the system architecture design have been presented we are ready to examine a prototype implementation. Subsequent evaluation of the prototype will provide insight into the validity of this solution to the task of distributed, interactive, VE simulation.