A Software Architecture for Modeling and Distributing Virtual Environments

Rycharde Hawkes

Doctor of Philosophy
The University of Edinburgh
1996
In memory of

Martyn “Rocket” Howard
Abstract

The simulation of a Virtual Environment (VE) is an intensive process which is severely limited if restricted to one machine. Through distribution it is possible to increase the size and accuracy of the simulation, thus permitting multiple users to interact with each other and the VE.

Existing distributed VE systems have been designed to target a specific level of distribution. This level is dictated by the geographical distance over which the systems must operate and the communications medium connecting them. The system requirements on a tightly-coupled multiprocessor system are not the same as those of a system operating over a Wide Area Network (WAN). Consequently, the solution for any given level does not scale well to larger or smaller system configurations.

VE modeling has its heritage in Computer-Aided Design (CAD) and has evolved unchecked into its present state. As the amount of information required in a VE increases, so the current modeling techniques and tools are put under added stress to cope with the extra load. Most modeling techniques are driven by the structure of the system upon which the model must execute, rather than capturing the structure of the information it should represent.

This thesis questions the motives behind VE modeling, examines the problems of distributing a VE and details the various solutions that have been employed. An analysis of the methods used leads to the selection of techniques which may be combined to provide a solution unified over all levels of distribution. The proposed solution is also integrated with and actively supports the modeling process, thus providing a powerful environment for VE designers and participants alike.

The architecture of this system is presented complete with a description of a prototype implementation that demonstrates the key aspects. The thesis concludes with an evaluation of the prototype.
Acknowledgements

There are many people that I would like to thank, but this thesis is long enough already. Therefore I would like to specifically mention just a few: my supervisor, Eric McKenzie; Martin Reddy and David Stephenson for actually reading the whole thesis; Pierre duPont for answering many questions; my friends and family for their understanding and encouragement over the years; my invisible friend Bobo the pink flying elephant, who prevented me from losing my mind; and last, but not least, Frederike van Wijck, without whose support I would surely have never finished.

This thesis was undertaken on a part-time basis whilst the author was employed on the JCI/MRC Grant G9212693: Visual Control of Steering.
Declaration

I declare that this thesis was composed by myself, and that the work contained therein is my own, except where explicitly stated otherwise in the text.

Rycharde Hawkes
Contents

1. INTRODUCTION .................................................................................................................... 1
   1.1 MOTIVATION .................................................................................................................. 2
   1.2 A MODELING TIME-LINE ............................................................................................ 3
      1.2.1 Past ....................................................................................................................... 3
      1.2.2 Present ................................................................................................................ 4
      1.2.3 Future .................................................................................................................. 5
   1.3 DISTRIBUTING SIMULATIONS .................................................................................... 6
   1.4 INTERACTIVITY .......................................................................................................... 6
   1.5 THESIS PREVIEW ..................................................................................................... 8
   1.6 SUMMARY .................................................................................................................. 9

2. DESIGN ISSUES FOR DISTRIBUTED VIRTUAL ENVIRONMENT SYSTEMS .......... 10
   2.1 DISCRETE EVENT SIMULATION HERITAGE ........................................................... 11
      2.1.1 Optimistic versus Conservative ............................................................................ 11
      2.1.2 Time Warp .......................................................................................................... 12
      2.1.3 Discrete Event Simulation Summary .................................................................... 13
   2.2 ISSUES ....................................................................................................................... 14
      2.2.1 Real-time .............................................................................................................. 14
      2.2.2 Communications ................................................................................................. 15
      2.2.3 Data Management .............................................................................................. 18
      2.2.4 Computation Management .................................................................................. 22
      2.2.5 VE Modeling ....................................................................................................... 25
      2.2.6 Time Management .............................................................................................. 27
      2.2.7 Fault Tolerance .................................................................................................. 28
      2.2.8 Security ............................................................................................................... 30
      2.2.9 Issues Summary .................................................................................................. 30
   2.3 IMPLEMENTATIONS .................................................................................................. 31
      2.3.1 SIMulation NETworking System (SIMNET) ......................................................... 31
      2.3.2 Distributed Interactive Simulation (DIS) ............................................................... 32
      2.3.3 Naval Postgraduate School Networked Vehicle Simulator IV (NPSNET-IV) ......... 35
      2.3.4 Minimal Reality (MR) Toolkit .............................................................................. 39
      2.3.5 Distributed Interactive Virtual Environment (DIVE) .......................................... 42
      2.3.6 Distributed Virtual Environment System (dV S) ................................................ 45
      2.3.7 Waterloo Virtual Environment System (WAVES) .............................................. 48
      2.3.8 AVIARY ............................................................................................................... 51
      2.3.9 A Universal Architecture ................................................................................... 54
   2.4 SUMMARY .................................................................................................................. 54
      2.4.1 Communication Organisations ............................................................................. 54
      2.4.2 Transport Mechanisms ....................................................................................... 56
      2.4.3 Bandwidth Implications ...................................................................................... 56
      2.4.4 Distribution & Scaleability ................................................................................ 57
      2.4.5 Time .................................................................................................................... 60
      2.4.6 Fault Tolerance .................................................................................................. 61
      2.4.7 Security .............................................................................................................. 61
      2.4.8 Modeling ............................................................................................................ 62
      2.4.9 System Summaries ............................................................................................ 63
      2.4.10 A Universal Architecture ................................................................................. 64
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.3 Sun SPARCcenter</td>
<td>177</td>
</tr>
<tr>
<td>5.2.4 SGI RealityStation</td>
<td>177</td>
</tr>
<tr>
<td>5.2.5 Implementation Language</td>
<td>177</td>
</tr>
<tr>
<td>5.3 NETWORKING</td>
<td>178</td>
</tr>
<tr>
<td>5.3.1 IPC Mechanisms</td>
<td>179</td>
</tr>
<tr>
<td>5.3.2 Addresses</td>
<td>180</td>
</tr>
<tr>
<td>5.3.3 Messages</td>
<td>181</td>
</tr>
<tr>
<td>5.3.4 Hardware Differences</td>
<td>182</td>
</tr>
<tr>
<td>5.3.5 Layer Implementation</td>
<td>185</td>
</tr>
<tr>
<td>5.3.6 Networking Summary</td>
<td>190</td>
</tr>
<tr>
<td>5.4 CONFIGURATION CONTROL</td>
<td>190</td>
</tr>
<tr>
<td>5.4.1 Universal Configuration Language (UCL)</td>
<td>191</td>
</tr>
<tr>
<td>5.4.2 System Configuration</td>
<td>192</td>
</tr>
<tr>
<td>5.5 A UML INTERPRETER</td>
<td>193</td>
</tr>
<tr>
<td>5.5.1 Overall Structure</td>
<td>193</td>
</tr>
<tr>
<td>5.5.2 Interpreting the Data Definition</td>
<td>194</td>
</tr>
<tr>
<td>5.5.3 Instancing</td>
<td>195</td>
</tr>
<tr>
<td>5.5.4 Component Dependencies</td>
<td>199</td>
</tr>
<tr>
<td>5.5.5 Interpreting Instruction Code</td>
<td>200</td>
</tr>
<tr>
<td>5.5.6 Interpreter Embedding</td>
<td>202</td>
</tr>
<tr>
<td>5.5.7 Persistence</td>
<td>203</td>
</tr>
<tr>
<td>5.6 UNIVERSE MANAGER</td>
<td>203</td>
</tr>
<tr>
<td>5.6.1 Node Initialisation</td>
<td>204</td>
</tr>
<tr>
<td>5.6.2 System Initialisation</td>
<td>204</td>
</tr>
<tr>
<td>5.6.3 Managing Processes</td>
<td>207</td>
</tr>
<tr>
<td>5.6.4 Processing Service Requests</td>
<td>209</td>
</tr>
<tr>
<td>5.6.5 Entity Migration</td>
<td>210</td>
</tr>
<tr>
<td>5.6.6 System Interaction</td>
<td>211</td>
</tr>
<tr>
<td>5.6.7 System Termination</td>
<td>212</td>
</tr>
<tr>
<td>5.7 RESOURCE MANAGER</td>
<td>212</td>
</tr>
<tr>
<td>5.7.1 Resource Consumption</td>
<td>212</td>
</tr>
<tr>
<td>5.7.2 Initialisation</td>
<td>213</td>
</tr>
<tr>
<td>5.7.3 Services</td>
<td>214</td>
</tr>
<tr>
<td>5.8 ENTITY LIBRARY</td>
<td>215</td>
</tr>
<tr>
<td>5.8.1 Initialisation</td>
<td>215</td>
</tr>
<tr>
<td>5.8.2 Service Requests</td>
<td>215</td>
</tr>
<tr>
<td>5.9 MANAGER LIBRARY</td>
<td>216</td>
</tr>
<tr>
<td>5.9.1 Initialisation</td>
<td>216</td>
</tr>
<tr>
<td>5.9.2 Simulation Loop</td>
<td>217</td>
</tr>
<tr>
<td>5.10 VISUAL MANAGER</td>
<td>218</td>
</tr>
<tr>
<td>5.10.1 Initialisation</td>
<td>219</td>
</tr>
<tr>
<td>5.10.2 Simulation Loop</td>
<td>219</td>
</tr>
<tr>
<td>5.10.3 Entity Enhancement</td>
<td>219</td>
</tr>
<tr>
<td>5.10.4 VIS Summary</td>
<td>220</td>
</tr>
<tr>
<td>5.11 CONSOLE</td>
<td>220</td>
</tr>
<tr>
<td>5.12 FURTHER IMPROVEMENTS</td>
<td>220</td>
</tr>
<tr>
<td>5.12.1 Configuration</td>
<td>220</td>
</tr>
<tr>
<td>5.12.2 Multi-part Messages</td>
<td>221</td>
</tr>
<tr>
<td>5.12.3 State Encoding</td>
<td>221</td>
</tr>
<tr>
<td>5.12.4 Persistence</td>
<td>221</td>
</tr>
<tr>
<td>5.12.5 Message Elimination</td>
<td>222</td>
</tr>
<tr>
<td>5.12.6 Entity Synchronisation</td>
<td>222</td>
</tr>
<tr>
<td>5.12.7 Function Access</td>
<td>222</td>
</tr>
<tr>
<td>5.13 SUMMARY</td>
<td>223</td>
</tr>
</tbody>
</table>
List of Figures

FIGURE 2.1 EXAMPLE INTERACTION MATRIX ................................................................. 25
FIGURE 2.2 A CLOCK SYNCHRONISATION FORMULA ............................................... 28
FIGURE 2.3 DIS PERFORMANCE WITH DIFFERENT DEAD-RECKONING ACCURACIES 35
FIGURE 2.4 MR TOOLKIT COMPONENT STRUCTURE .................................................. 40
FIGURE 2.5 DVS SYSTEM ARCHITECTURE .................................................................. 45
FIGURE 2.6 BASIC WAVES ARCHITECTURE ............................................................. 49
FIGURE 2.7 AVIARY COMPONENT SCHEMATIC .......................................................... 52

FIGURE 3.1 CONFECTION OF ARTIFICIAL AND REAL ENVIRONMENTS ................ 72
FIGURE 3.2 TYPE OF TIME AND SPACE ....................................................................... 72
FIGURE 3.3 FUNDAMENTAL ELEMENTS OF A VIRTUAL ENVIRONMENT ................ 75
FIGURE 3.4 A CLASSIFICATION SCHEME FOR VIRTUAL ENVIRONMENTS .......... 75
FIGURE 3.5 UNIVERSE HIERARCHY TREE SHOWING POSSIBLE ENTITY MIGRATION PATHS 82
FIGURE 3.6 THE EFFECT OF UPDATE RATE ON TIME-TO-CONTACT (TTC) .............. 88

FIGURE 4.1 A TAXONOMY OF REAL-TIME SCHEDULING ALGORITHMS .................. 101
FIGURE 4.2 BACKUS-NAUR FORM DESCRIPTION SHOWING RELATIONSHIPS BETWEEN UML COMPONENTS ................................................................. 118
FIGURE 4.3 EXAMPLE TOP-LEVEL UML DESCRIPTION ........................................... 119
FIGURE 4.4 A POSSIBLE DEFINITION FOR THE VISUAL ELEMENT ....................... 121
FIGURE 4.5 DEFINING A UNIVERSE BY INHERITANCE ............................................ 122
FIGURE 4.6 INHERITING FROM AN ELEMENT .......................................................... 123
FIGURE 4.7 EXPLICIT/IMPLICIT INVOCATION OF A CONVERTER ....................... 123
FIGURE 4.8 DEFINITION OF AN ENTITY ..................................................................... 124
FIGURE 4.9 METHODS FOR ACCESSING MEMBER PROPERTIES IN ELEMENTS ........ 126
FIGURE 4.10 USER AND SYSTEM FUNCTION CALL EXECUTION .................................. 127
FIGURE 4.11 ATTEMPTING TO CONVERT AN ELEMENT WITHOUT A CONVERTER 128
FIGURE 4.12 CALCULATING A TIME DELTA USING STATE INDEXING ...................... 128
FIGURE 4.13 IMPORTING A MODULE ........................................................................ 129
FIGURE 4.14 EXAMPLE STRUCTURE OF A UNIVERSAL SIMULATION SYSTEM .... 131
FIGURE 4.15 EXAMPLE ORGANISATION OF A USS COMPLETE WITH POPULATED USNS 133
FIGURE 4.16 HIERARCHICAL STRUCTURING OF USSS ............................................. 136
FIGURE 4.17 ORDER OF EVENTS FOR A SIMULATION UPDATE ............................... 139
FIGURE 4.18 POSSIBLE COMMUNICATION PATH TAKEN BY A MESSAGE SENT FROM AN ENTITY TO ALL MANAGERS ................................................................. 140
FIGURE 4.19 PROCEDURE FOR REGISTERING INTEREST IN A UML COMPONENT .... 141
FIGURE 4.20 SEQUENCE OF EVENTS DURING ENTITY CONSTRUCTION ................ 144

FIGURE 5.1 SIMULATION CYCLE SCHEDULING ......................................................... 171
FIGURE 5.2 PSEUDO-CODE FOR THE FIXED FRAME RATE, WORST-CASE SIMULATION CYCLE 172
FIGURE 5.3 IMPROVED SIMULATION CYCLE SCHEDULING .................................... 173
FIGURE 5.4 EXAMPLE NETWORK CONFIGURATIONS OF THREE USSS ................. 178
FIGURE 5.5 POSITION OF THE PROCESS MANAGEMENT LAYER WITHIN THE SYSTEM SOFTWARE 180
FIGURE 5.6 SEND-RECEIVE-REPLY PROCEDURE FOR SENDING MESSAGES UNDER QNX 186
FIGURE 5.7 MESSAGE TRANSMISSION SEQUENCE USING TCP/IP .......................... 188
FIGURE 5.8 STRUCTURE OF A LOGICAL PROCESS CONSISTING OF TWO PHYSICAL PROCESSES 188
FIGURE 5.9 THE BASIC ELEMENTS OF UCL ............................................................ 191
FIGURE 5.10 EXAMPLE USS CONFIGURATION FILE .................................................. 192
List of Tables

TABLE 2.1 KLEINROCK DISTRIBUTION CLASSIFICATION SCHEME ....................................................... 19
TABLE 2.2 DEGREES OF FAULT TOLERANCE ..................................................................................... 29
TABLE 2.3 DISTRIBUTED VE SYSTEM FEATURE CLASSIFICATION SUMMARY ........................................ 55

TABLE 3.1 COMMON SENSES AND THEIR SENSITIVITY/RESOLUTION ..................................................... 69
TABLE 4.1 NETWORKING MEDIUM PROPERTIES ................................................................................. 99
TABLE 4.2 INTERPRETER RESOURCE EVALUATION .......................................................................... 114

TABLE 5.1 MESSAGE ADDRESS STRUCTURE ..................................................................................... 180
TABLE 5.2 MESSAGE HEADER STRUCTURE ..................................................................................... 181
TABLE 5.3 DESCRIPTION OF THE NINE PHYSICAL MESSAGE STRUCTURES .......................................... 182
TABLE 5.4 SUMMARY OF MESSAGE TYPES AND THEIR USE ................................................................. 183
TABLE 5.5 SAMPLE ENTITY DISTRIBUTION OVER THREE NODES .................................................... 205

TABLE 6.1 RESOURCE RATINGS FOR EACH TEST PLATFORM ............................................................. 228
TABLE 6.2 MINIMUM MEMORY USAGE OF USS COMPONENTS ........................................................... 231
TABLE 6.3 EFFECTS OF TECHNIQUES TO REDUCE CODE SIZE ............................................................ 234
TABLE 6.4 MEMORY CONSUMPTION OF THE FOUR PRIMITIVE UML TYPES ........................................ 235
TABLE 6.5 APPROXIMATE MEMORY USAGE FOR UML COMPONENTS .................................................. 238
TABLE 6.6 EXAMPLE OF HOW MUCH MEMORY IS ALLOCATED TO REPRESENT A UML DEFINITION AND HOLD ITS INSTANCE DATA UNDER QNX ................................................................. 239
TABLE 6.7 FUNDAMENTAL INTERPRETER OPERATIONS TIMINGS FOR EACH TEST PLATFORM .............. 240
TABLE 6.8 OPERATION OVERHEADS PER INTEGER ARRAY ELEMENT .................................................. 243
TABLE 6.9 TIME PENALTIES INCURRED WHEN RM MONITORS CPU USAGE ...................................... 281

TABLE 7.1 COMPARISON OF DISTRIBUTED VE FEATURE CLASSIFICATIONS INCLUDING USS ............ 302
Chapter 1

Introduction

"Research is what I'm doing when I don't know what I'm doing."

Wernher Von Braun

The subject matter of this thesis falls in the area commonly described as Virtual Reality (VR). Ask anybody to describe what VR is and you will get a different answer. The term was originally coined by Jaron Lanier to describe a system using immersive technology, such as Head-Mounted Displays (HMDs) and data gloves (Pimentel and Teixeira, 1993). Since then the perception of what VR is has changed, for better or worse, to encompass many different combinations of novel (and not so novel) input and output devices. The common factor between all of these is the use of three-dimensional (3D) computer graphics. The layman would therefore be forgiven for thinking that anything that uses 3D graphics is VR - a connection often reinforced by the media.

The term Virtual Environment (VE) is used to describe the environment that one enters when using a VR system. This has also become popular but it is an inaccurate description because there is nothing virtual about the environment (this topic is dealt with later). Essentially VR is used to refer to the whole subject area, its hardware, software, applications, etc., and a VE is the thing being partly or wholly simulated by the VR system.

This brief introduction describes the author's motivation behind the work presented in this thesis. The next section outlines the author's perspective on why VEs are modeled
the way they are presently, what should change and why distribution is necessary. This chapter concludes with a preview of the contents of this thesis.

1.1 Motivation

The author first became interested in the field of VR in 1991 whilst working for a company that built real-time 3D Computer Image Generators (CIGs). There were two stages required to build an application using these CIGs: modeling and coding. First of all the geometrical and surface properties, i.e. colour, texture, etc., of the 3D objects that would populate the simulated environment were described in a special 3D modeling package. These were then converted into the CIG’s native model format and their behaviour coded into the main body of the application. The variables needed to describe the objects’ behaviours depended on the nature of the simulation. Some objects would be under user control and thus behave as the user wanted. The behaviours of the computer controlled objects were often choreographed to obtain the best visual effect. This was usually achieved by breaking down the movements into a series of parameterised actions which were called in sequence to effect the desired behaviour.

Each time a simulation was developed, as many existing models as possible were recycled and organised with new purpose-built models in the standalone modeling package. Traditionally the application code was written again from scratch, except for a few key routines. After having used this process a couple of times, the author designed a core extensible application framework that could be specialised for each simulation. Although the properties of objects could be encapsulated and reused where possible, they were still held and manipulated separately to their geometrical representations.

After joining the Virtual Environment Laboratory (VEL) at the University of Edinburgh a year later, the need for a flexible system to model and support VEs became even more apparent. The visual perception experiments undertaken in the laboratory required many varied environments. Often these were modified slightly for
various trials to provide a basis for comparison of the user's performance. Both immersive and non-immersive displays were used, complemented with appropriate input devices. The target platform for the system was a network of IBM Personal Computer (PC) compatibles. Due to the large number of devices and tasks that were required to simulate the VEs, the devices and simulation workload had to be distributed amongst these machines.

The design of the system was constrained by the technology used and it was at this point that the concepts underlying a more ideal architecture began to form. This thesis represents the development of these initial ideas into a coherent design and a prototype implementation for a system capable of modeling and executing VEs on different types of machines connected over varying distances.

1.2 A Modeling Time-Line

To understand the VE modeling techniques used today, it is beneficial to look at the heritage that has influenced the current process. With this knowledge we may reflect on existing approaches and speculate on how these will (or should) change in the future.

1.2.1 Past

The strong relationship that has been established between VR and visuals is not an accident. Pictures drawn by computers have fascinated people for the past three decades and, shortly after this ability was recognised, they were applied to a real world task. Computer-Aided Design (CAD) started its life as a two-dimensional electronic technical drawing bench and has, over the years, naturally progressed into the third dimension. Initially models were pure geometry, but as the applications of CAD increased in step with processing power, so other attributes were added such as material properties. Amongst the properties described were the material's visual appearance, e.g. colour, texture, reflectivity, etc. Nowadays high quality renderings
representing realistic materials can be produced from CAD models which are used to design everything from bolts to skyscrapers.

Whilst one branch of computer graphics worked on attaining realism, another concentrated on speeding the process up so that interaction with the images was possible. The military were one of the first institutions to recognise the possible applications for real-time 3D graphics and they had the money required to fund the development of the necessary hardware and software. The resulting spectrum of solutions covered the high-end, high quality flight simulators (Schachter, 1981) down to the (relatively) low-cost SIMNET networked tank simulators (Kanarick, 1991). These simulators were built around a fast visual display but now there was also a requirement to model additional information. Not just material properties, but the attributes of the actual thing being simulated which, by necessity, also included its environment. This extra information was typically specified separately from the visual model of the simulation and both were managed simultaneously by the simulator software.

1.2.2 Present

The birth of VR signalled the start of a reintegration of the various areas of computer graphics. Technology was sufficiently advanced and at a price which meant that such systems were affordable by more people. One of the earliest applications was architectural walk-throughs which presented CAD models at interactive rates (Airey et al., 1990). The line between the VR and low-end real-time simulation markets has also become blurred and, for the most part, has meant absorbing the complexity of the simulations.

Audio is now rated favourably with visuals and sound effects are not limited to plain stereo but may be positioned and oriented in 3D space (Wheeler et al., 1993). Single projection displays have been joined by many types of stereoscopic displays which present a pseudo-3D view on the VE (Rushton and Wann, 1993). There is active research into tactile displays which are dependent on surface textures and their
properties for the technology's success (Minsky et al., 1990), e.g. softness, apparent temperature, etc. Force-feedback devices have also been used in applications, the most cited of which is molecular docking (Ouh-young et al., 1988). Subsequently, there is a need for Physically-Based Modeling (PBM) of the VE which can rely on a considerable number of variables and equations. Of course there is no requirement to develop VEs that closely model our own environment, which means the structure and content of the information accompanying the seemingly obligatory visuals can vary a great deal. Indeed, it may be beneficial to model information that is not part of the VE per se but affects how objects interact within it, e.g. medium, aura, focus, nimbus and adapters (Benford and Fahlén, 1993).

Attempting to meet this sudden increase in information, existing visual modelers have been retrofitted with new features to accommodate some of the non-visual information that designers want to model, e.g. audio links, behavioural information, physically-based modeling parameters, etc. The result is often unwieldy and inflexible with modeling still centred around visuals instead of approaching the modeling task without bias. This is, in fact, the best case; it is still common to find integration of data within the application rather than at a higher-level. This is partially due to the fact that each VE system uses its own modeling system with a proprietary structure and format. Certainly any exchange of information requires an explicit conversion process which can often lead to a loss of detail and/or a sub-standard content.

1.2.3 Future

The amount and type of information that needs to be modeled will inevitably increase and, unless a suitable flexible framework is adopted, the VE model may collapse under its own weight. Standardisation of any area is generally a bad idea when that area is not well understood, but if each proposed solution is sufficiently flexible then there is the possibility of a gradual merging until, eventually, only one form exists. This approach can be applied to VE modeling which can take advantage of the benefits of standardisation to aid high-level tools development and ease data exchange. A good starting point for the development of such a model would be the elimination of the
emphasis on any one type/medium of information used to build a VE, e.g. visuals, audio, etc.

1.3 Distributing Simulations

The more complex a model becomes, the more computing power a system will need to execute it. Only so much computational power can be squeezed into a single machine and, for anything other than small models, it will be necessary to distribute the simulation between a number of machines to cope with the extra load. In this way more efficient use of each machine's resources can be made and the possibility of multiple user interaction is introduced.

The problems of distributing a simulation over a number of machines are many and are compounded by increasing the distance between machines. These problems are slightly different depending on the combination of hardware used and the geographical distance covered. There is no one technique that can be applied at all levels of distribution that will address all of the problems posed. Therefore a suitable multi-level solution is needed that applies the right technique in the right place.

Ideally, the modeling technique should influence the architecture of the simulation system but it is not uncommon for this situation to be reversed (DIS, 1994). If improvements are to be made to the modeling process, it is essential that the underlying system provides the comprehensive support necessary.

1.4 Interactivity

The work presented in this thesis first takes a broad look at VE systems and then concentrates on a specific aspect: interactivity. The adjective "interactive" is commonly used to indicate that the thing it is applied to runs at a fast enough rate to form some relationship with the human user. Many of the observations and techniques described in this work are valid regardless of the applications such a system is applied to, but, in light of the primary concern, the emphasis has been placed on two factors: consistency and real-time.
Consistency refers to the problem of ensuring that the VE each participant is interacting with appears the same in spite of the fact that it may be distributed over a number machines covering a certain geographical distance. It also deals with the issues regarding multiple users and the problems they bring, e.g. two users may not simultaneously manipulate the same properties of a given object.

Whilst interactivity is a goal, "real-time" identifies a set of techniques that may be used to realise that goal. The latter term is often confusingly used to describe interactive systems as the author will find in chapter 2 when current VE systems are reviewed. However, the author has attempted to distinguish the two starting in chapters 3 and 4. Consequently, real-time has been applied in two ways to the original work in this thesis. Firstly, to describe real-time displays that produce a fast, constant update rate to enable effective interaction; and secondly, to describe the fundamental nature of the system that permits these types of displays to be realised and support consistency.

Real-time displays are a requirement of the ideal VE system considered here, but are not essential for all the applications that such a system may be used for. For example, somebody visualising a complex data set may be happy to tolerate a few display updates a second, whereas a pilot in a flight simulator may find his job very difficult if the display is updated less than 60 times a second. These examples may also be used to scope the importance of consistency. Modification of one part of the data set whilst another person views a different portion may be perfectly acceptable if it does not affect that person's task. On the other hand, suddenly introducing another plane into a networked flight simulation or perhaps removing part of the terrain could have quite profound consequences.

It is very important to understand that a real-time display is not a physical display that is just updated fast, e.g. a monitor, it is a display of the VE which itself is updated at a fast rate with a constant duration in between updates. Possible types of displays include visual, aural and tactile.
1.5 Thesis Preview

Chapter 2 presents a method of classifying the issues involved in the design of a system capable of distributing VEs. Existing solutions to this problem are described, including their approaches to modeling, and then compared using the classification scheme.

Chapter 3 looks at the whole concept of environment modeling, reassesses what we are trying to accomplish and presents a new approach to the task. During this process, the structure of our natural environment is examined in the hope that it will provide enlightenment about modeling in general. This section concludes by deriving a suitable definition and abstract model for a VE. Finally, an aspect of human-computer interaction is highlighted which has implications on how VEs are simulated. Many systems today have variable-rate displays that distort some of the information a human uses to make decisions. A visual perception theory is used to demonstrate how a constant-rate display can resolve this problem.

Based upon the knowledge gained in the previous chapters, the design of a new distributed VE system is presented in chapter 4. First of all, a flexible modeling language is described that is integral to the system architecture. Rather than targeting a specific set of hardware or geographical distance, the system solution is structured in such a way that the correct techniques are applied at the right time, so that all configurations may be supported.

The implementation of a prototype system is described in chapter 5. Not all of the design's elements are fully implemented, but it is sufficiently represented to verify the viability of the ideas used in the proposed solution. Each of the core system components are dealt with in turn, addressing the key decisions taken during implementation and the major data structures used.

Chapter 6 is an evaluation of the prototype which was implemented on a number of test platforms. Performance of the building-block components is established before dealing with system performance as a whole. The chapter concludes by outlining a number of
enhancements that could be made to the design and implementation in order to improve the prototype's performance.

The thesis concludes in chapter 7 with the application of the classification scheme to the proposed system, a summary of its most important features and suggestions for further work.

1.6 Summary

This chapter began with a cursory introduction to VR and VEs which is significantly expanded in the next two chapters. The author's motivations for this work were based purely on practical experience, combined with the wish to make the development of and interaction with VEs less painful. The reasons for the current state of VE modeling were outlined and their weaknesses exposed. At the centre of any solution to this problem is the modeling system. A more flexible approach is required, as well as the underlying framework to support this process and an integrated modeling/simulation system capable of handling the result. The road to a new software architecture begins with an examination of existing system solutions.
When looking at existing systems we are interested in their solutions to two problems: how they tackle the problem of VE modeling and how they "execute" a given VE. The former is a much more abstract area and in theory may be independent of the underlying mechanism of distribution and VE support software. However, in reality this is rarely the case. Sometimes the implementation drives the modeling system used and vice-versa. Whilst treating these aspects separately is desirable, it is also very difficult since describing one aspect cannot be done without referral to features of the other. This chapter examines the issues that must be addressed when designing a VE system. Existing distributed simulation systems solutions are analysed with reference to the outlined issues and comparisons are drawn.

The term "distributed simulation" is very general and is open to many interpretations. "Simulations" can be broadly classified as either off-line/computationally intensive or interactive/low fidelity. The first class is the type of simulation that is often called discrete event simulation whereas driving and flight simulators would fall into the second class. A similar decomposition of "distribution" may also be attempted. It can be used to describe a simulation that is distributed over a number of tightly-coupled computational nodes with the intention to speed up the calculations. This fits well with the first class of simulation and when considering VR and interactive simulations
this definition is also valid. However, the emphasis is more on the distribution of the simulation over some geographical distance such that multiple people may interact. Each of these types have their own requirements and hence their solutions cannot necessarily be applied to each other's problems. For example, the fact that there is a human being interacting with the simulation brings onboard a number of new requirements or, more realistically, constraints on how the simulation may behave.

2.1 Discrete Event Simulation Heritage

Before re-inventing the wheel it is beneficial to look at the historically largest form of simulation: discrete event simulation. There are two approaches for ensuring the correctness of a distributed simulation\(^1\): optimistic and conservative.

2.1.1 Optimistic versus Conservative

Initially, all simulations used a conservative solution to control their progression. Each simulation consists of a certain number of processes. Only when all processes have completed their work will simulation time increase and the next cycle commence. The obvious disadvantage to this approach is that those processes that take significantly less time to complete their work will be forced to wait. If each process was allocated to a physical processor then this would result in a considerable waste of the computational resources.

To overcome this weakness a different approach was sought. The optimistic solution permits each process to progress at their own rate. This would work fine if all processes were independent of one another. Unfortunately this is often not true and a situation may arise where a slow process communicates with a faster process indicating that their previous work was in error. Since all of the fast process' subsequent work was based on an invalid state, this must be abandoned and recalculated using the correct state. The method to restore this state is known as

\(^1\) In discrete event simulation, distribution is almost always used just to increase the simulation's speed.
rollback. This solution is called optimistic because it works on the assumption that the situations requiring rollbacks rarely occur.

2.1.2 Time Warp

Time Warp (TW) is an optimistic policy simulation model that is structured as a number of processes that each maintain a Local Virtual Time (LVT) (Jefferson and Sowizral, 1985). Each process may progress at its own rate, advancing LVT as necessary. Each message that is sent between processes indicates the LVT of the sender and is used to decide whether a rollback is required of the receiver. Keeping a list of what has happened in the past soon eats into the resources of each process, so a mechanism for collecting old data has been provided.

At periodic intervals, the operating system interrogates each process for their LVT. Then the system's Global Virtual Time (GVT) is updated to show the progression of the simulation, taking into consideration the slowest process. When GVT is updated, any data previous to this time may be discarded since rollback may not occur before GVT. The choice of algorithm to calculate GVT is crucial to system performance and can make the difference between running or not running a simulation if large state lists are required (Bellenot, 1990; D'Souza et al., 1994).

Further optimisation may be made by finding a way to reduce the amount of state saved in these lists. A basic mechanism would save the complete process state, however this is expensive both in terms of time taken to save the state and the time taken to perform a rollback. By performing incremental state saving (Cleary et al., 1994), i.e. only saving the state that has changed, it is possible to improve efficiency. An alternative approach, called adaptive checkpointing, is to adjust the rate at which the process state is saved based upon the rollback behaviour (Rönngren and Ayani, 1994).
2.1.3 Discrete Event Simulation Summary

TW works well in discrete-event simulations and is a very popular model, but there are a number of problems. TW was not designed to be used for interactive applications which rely on completing all computations in a very small amount of time (~33 ms to achieve a 30 Hz update rate). In order to ensure that these strict deadlines are met, some notion of predictability must be provided. Rollback is a result of processes being allowed to continue at their own rate and can be seen as self defeating since the rate of progress is not controlled and the occurrences of rollbacks are unpredictable. In addition, one rollback may trigger another rollback in another process and so on until, potentially, each process has been rolled back to GVT.

However, there has been some recent work on the application of TW to real-time simulations resulting in the development of a Parallel Optimistic Real-Time Simulator (PORTS - Ghosh et al., 1994). In PORTS, GVT is calculated continuously, i.e. after each event in the simulation, in order to speed the commitment of I/O operations. Incremental state saving is shown to be unpredictable and one proposed solution is to save the complete state every \( n \) events (where \( n \) is a constant for a particular simulation) in a similar way to adaptive checkpointing. This enables a bounded value for state saving and state restoration, thereby having predictable properties in the simulator. Deadline scheduling is also simplified because there is no event-migration or explicit load balancing and is done on a per-processor basis.

Despite this encouraging work, the application of PORTS to interactive simulations is unlikely. Take the case of a driving simulator where the driver is monitoring the environment and taking actions accordingly. Any rollbacks could interrupt the flow of time and would make it seem as though they are being controlled like a video recorder - pause, rewind, fast-forward and play - clearly defeating the goal of realism. In short, you cannot rollback a human being.

Conservative solutions ensure that situations that would require a rollback do not happen at all by, what proponents of optimistic policy would see as, restricting the progress of the simulation. This has the potential to under-utilise the available
resources, but with a good load-balancing algorithm the impact of such an approach can be reduced. The perceived advantage of an optimistic mechanism is that if a process requires very little interaction with other processes in the simulation, faster progression may be made if it is allowed to go at its own rate (Lipton and Mizell, 1990). This may also be perceived as a waste of resources that may be better allocated to other processes in the simulation.

Therefore, since there may be many humans interacting with the simulations of VEs, a conservative system is the only workable solution. This will also aid predictability and scheduling to meet the real-time deadlines that are required of a VE system (discussed in section 3.3).

2.2 Issues

There are many problem areas to consider when building a VE system and there are even more implications. There is no established classification scheme available with which these areas can be examined and different solutions compared, so an attempt has been made to construct one. Separating one area from another was more difficult in some cases than others. Not breaking a problem area down into separate issues would make comparison difficult, on the other hand, splitting the area into too many issues would provide a distorted representation. There are a lot of interdependencies between these issues, but it is hoped that the divisions made will aid comparison rather than hinder comprehension. This section looks at each issue in turn and assesses the impact they have on system design.

2.2.1 Real-time

The largest single constraint on an interactive simulation is that it must operate in real-time. As described in section 1.4, a real-time system permits the generation of real-time displays which are updated fast enough to allow the participant to effectively interact with the simulation and other participants. How fast may vary depending upon the exact nature of the simulation, but the goal is to reduce the delays between human action and simulation reaction to an imperceptible constant duration.
A simulation is composed of a sequence of discrete time steps in between which the calculations to update the environment must be completed. Failure to achieve this could result in a breakdown of realism (if that is being striven for) or, at the very least, a reduction in the efficiency of the participant to interact with the simulation. While it is true that simulation time may continue at any rate if there is no human or time-dependent device involved in the loop, we are primarily interested in interactive simulations and therefore the actual time between each simulation time step must be constant. We live in a constant world and to require us to interact with anything other than this is contrary to all our natural skills and will present us with corresponding difficulties (Hawkes et al., 1995). This is discussed further in section 3.3.

If these stringent deadlines are to be met then there must be a degree of predictability in the simulation's execution. An optimistic solution, as discussed earlier, is not very predictable whilst a conservative approach may be seen as a good basis to build upon. The design implications of real-time systems are discussed in section 4.3.

2.2.2 Communications

The structure of the communications subsystem is usually the most inflexible component of any system. The choice of platform and its location dictate what communications hardware is available. Consequently, the technique used to manage data is often directly influenced by this component.

2.2.2.1 Point-to-point

A point-to-point\(^2\) transfer of information may be achieved by either establishing a link between sender and receiver at every transmission, or creating a permanent connection which is destroyed when there will be no more communications. Connection-oriented protocols such as Transmission Control Protocol/Internet

\(^2\) Also known as unicast.
Protocol (TCP/IP) are commonly used and provide a reliable service. Unfortunately, ensuring that the receiver gets all the information and in the right order generates a fair amount of overhead. Furthermore, each receiver must acknowledge receipt of the transmission.

2.2.2.2 Broadcast

One alternative is to “broadcast” the information on the network and hope that anyone interested in that information will hear the broadcast and pick it up. This is the exact opposite of the point-to-point mechanism and is supported in the User Datagram Protocol (UDP). This connectionless protocol uses self contained, addressed packets (or datagrams) which puts the onus on the application to ensure that the data is processed in the correct order. The major advantage of this method is that there is no need to maintain a large number of connections. Apart from being unreliable, its main disadvantage is that it is possible to flood a network with broadcast messages which are of no interest to other connected systems and thus degrade performance.

2.2.2.3 Multicast

An improvement on broadcasting is multicasting. This works in the same way except that the packets are only sent to a subset of the network rather than the whole. Nodes may belong to one or more multicast groups and hence will only receive transmissions that are intended for them. It was originally available on LANs such as Ethernet and Fibre Distributed Data Interface (FDDI) but is now available at the network layer through the Multicast Backbone (MBONE - Macedonia et al., 1994). MBONE is a virtual network which runs on the same physical media as the Internet, but encapsulates multicast packets in normal IP packets and uses routers to forward them to their correct destinations. Multicast has yet to be standardised and consequently few implementations are available. More importantly, multicast per se is unreliable, although some research has been done on providing a reliable multicast service (Talpede and Ammar, 1995; Verissimo and Marques, 1990). However, unless
otherwise stated, any reference to multicast in this thesis is intended to describe the more common unreliable mechanism.

2.2.2.4 Bandwidth

The amount of data that may be transmitted in a given period of time has more impact on system design than any of these other factors. If only one network medium is being used then the task of designing an efficient protocol is relatively straightforward (but not simple). However, if multiple mediums are being catered for the problem becomes considerably more complex. A fast modem can manage approximately 28 Kbps, Ethernet has a bandwidth of 10 Mbps whilst FDDI and Fast Ethernet can offer 100 Mbps. It is quite common for this bandwidth to be shared amongst many other nodes thus reducing the effective data bandwidth considerably. There is also no way to guarantee a fraction of this bandwidth which adds to the problems. The evolving Asynchronous Transfer Mode (ATM) technology permits bandwidth to be reserved (channels), but this is currently even less available than multicast technology (Boisseau et al., 1995).

2.2.2.5 Latency

Communications latency is related to bandwidth and geographical distance. No matter what technological improvements are made, the speed of light will limit the transmission speed such that a latency of ~3 ms will be introduced for every 900 Km covered\(^3\). Thus design decisions are often based on the geographical distance over which the system will have to operate.

2.2.2.6 Shared Memory

This is a valid way of communicating between processes on the same node and the analogy can even be extended to operate over networks: distributed shared memory.

---

\(^3\) This calculation does not take into consideration the extra distance incurred as the light bounces off the interior of the optical fibre.
However, underlying such functionality is always some form of message passing. Bandwidth and latency can still be applied to shared memory. Whereas a message-passing system has built-in concurrency control, a shared memory system must add this itself, usually in the form of semaphores.

2.2.2.7 Structure

There are three commonly used models for communication in distributed VR systems: client/server, peer and hierarchical. In a client/server model one or more physical processes are designated as a server whose responsibility is to receive and process requests from clients for any of its published services. A client of one process can also be a server to another. This model works well for operating system resources, e.g. the filing system, network manager and process manager, where there is a limited number of potential clients and the client and server are tightly-coupled. If the number of clients gets too high, however, the server soon becomes a bottleneck.

The peer model essentially makes every process in the system equal in terms of functionality. This does not mean that there is any duplication of work between peers although this is quite common.

The hierarchical model uses a system whereby processes communicate with other processes in the hierarchy by sending the message to their parent process. The parent checks the address on the message and either sends it to one of its other children or to its parent process. This repeats until the message has arrived at its destination. Messages entering the hierarchy from outside are sent to the root (master) process which forwards the message as per normal. As with the client/server model, this master process may become a bottleneck if the number of child processes increases too far, or there is a large amount of communication with other process hierarchies.

2.2.3 Data Management

If the whole VE was managed by one machine then data management is straightforward, every process has direct access to the information they need with little overhead. If the VE is distributed across more than one machine then the
situation becomes more complex and requires a different solution. The overriding concern is to ensure that the integrity of the data is maintained at all times with minimal overhead. Other factors that affect solution selection are bandwidth and fault tolerance.

The nature of the target system and the geographical dispersion of the network dictates the type of management commonly used. All of the solutions currently offered fall within one of the categories shown in Table 2.1. Although general comparisons can be made between them, only those systems in the same category can be compared point for point.

<table>
<thead>
<tr>
<th></th>
<th>Tightly-Coupled</th>
<th>Loosely-Coupled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Near</td>
<td>Parallel Processing</td>
<td>Distributed Processing</td>
</tr>
<tr>
<td></td>
<td>High Speed LAN</td>
<td>LAN</td>
</tr>
<tr>
<td>Far</td>
<td>Impossible?</td>
<td>Distributed Access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WAN</td>
</tr>
</tbody>
</table>

LAN: Local Area Network
WAN: Wide Area Network

Table 2.1 Kleinrock distribution classification scheme.

2.2.3.1 Localisation

When the amount of data is small it is preferable that every process should have direct access to it. As the volume of data increases so does the burden on resources; memory and backing storage diminish rapidly and the amount of computation required to process the data rises dramatically. In a distributed system there is also an increase in network traffic as the data is moved around from one node to another.

It is therefore desirable to segment the data in some logical way such that any given process is only interested in one segment at a time (mostly). One common criteria used for segmentation is that of space. When the VE covers a large (virtual) distance it is broken up into a number of areas which are often allocated by and under the control of an Area Manager. The size of the areas can depend on many things, such as visibility, memory, speed of movement through the VE, etc., and upon the media,
e.g. visual, aural, etc. The shape of each area is often kept uniform for simplicity's sake. Rectangular areas are often favoured although some work has been done with hexagonal areas (Macedonia et al., 1995). However, some research has examined the subdivision of model space based on visibility alone (Airey et al., 1990). When applied to architectural models, the resulting binary space subdivision algorithm creates cells which are bounded by a number of splitting planes and can therefore be irregularly shaped.

2.2.3.2 Complete Distribution

This approach distributes the complete VE state between every node in the network. There is no duplication of information and any intention to change part of the VE state not under the control of a given process must be communicated to the process managing that data. Unless the state of the VE is distributed amongst all the nodes in the network sensibly, it is possible that such an arrangement could be detrimental to performance.

Since a given piece of data is only held in one place this solution is susceptible to machine failure or breaks in the communication paths. Such a technique can be applied at the near/tightly-coupled level and, perhaps, at the near/loosely-coupled level.

2.2.3.3 Partial Replication

When using partial replication only the parts of the VE state that will be modified by a given process will be held locally and only when needed. There are two sub-categories of partial replication: active and passive.

Active replication is where the process wishing to make the state change initiates the request for a local copy of the state. Modifications are made locally and the updated state is sent back to the originator.

Passive, or demand replication (Broll, 1995), requires an initial registration of interest in part (or all) of the VE state when the process is created. From that moment on it is
sent copies of that subset of state when it has been modified by one of the other processes. Changes may be made locally and sent back or, alternatively, the owner is informed of the desired changes and then makes them itself. A variation on this method is that the remote process receives updates of the object's exported behavioural model (section 2.2.4.5). In which case the remote process is not expected to want to modify the object's state, just monitor it.

If changes are made locally it would be possible for multiple copies to be taken from multiple processes, altered and submitted simultaneously, therefore resulting in an inconsistent state. To prevent this from happening a system of read/write locks may be employed. Before obtaining a local copy of the state for modification, a write lock is requested. This will be granted once any outstanding write locks are relinquished. Either the requester must block until the lock is granted, or a time-out can be specified which will permit the requester to continue with other work. On submitting the changes the modifications are made and the write lock released. If multiple locks need to be acquired before proceeding then the problem of deadlock also arises. There are several variations on this approach but all are equally complex. However, if all changes are made by the owner there is no need for this complex system and the modification process is a lot more predictable. This technique is most commonly used at the near/loosely-coupled level, although it could be applied at the far/loosely-coupled level if bandwidth was high enough.

2.2.3.4 Total Replication

This solution requires the complete VE state (or the essence of it) to be held at each node in the network. The two possible reasons for storing the complete VE state are, firstly, that the node's calculations are based upon most or all of that information or, secondly, the distance between nodes is so great that latency has become a real problem (far/loosely-coupled). This method does not scale well since every node must keep each other informed of updates which soon consumes bandwidth. The allocation of locks is infeasible so passive replication must be used to receive continuous updates on VE state.
2.2.4 Computation Management

Just as data is distributed, so can the computation. By computation we mean any work involving a specific object, whether it is an operation within or upon that object. Fortunately we can use similar categories to explore the options.

2.2.4.1 Complete Distribution

All operations on an object are performed on the same node that holds the object's data. If one process wishes to perform an operation on another then it must send a message to the other process. The allocation of processes to nodes may be optimised by enlisting the help of a load-balancing algorithm (section 4.3.1.4). By monitoring resource consumption and communication patterns the optimum allocation may be derived. This would permit most objects that often communicate with one another to be located on the same node - the movement of processes is commonly known as migration.

Such an approach works well on near/tightly-coupled systems but the latency and low bandwidth found in loosely-coupled systems can reduce its efficiency.

2.2.4.2 Partial Distribution

This method is similar to complete distribution except that the object's state is usually acquired using one of the passive or active partial data replication techniques and the changes made locally. There is, however, no duplication of computational effort.

2.2.4.3 Partial Replication

To compensate for slower communications links, e.g. near/loosely-coupled, it is possible to replicate some of the state computation on some or all of the nodes. These "ghost" or proxy processes are typically used to approximate the object's behaviour using a method called dead-reckoning (section 2.2.4.5). The process that performs the full simulation of that object also runs this model in parallel and when the two differ by a pre-defined amount, a copy of the real object's state variables is
sent to all of the ghost processes. Subsequent approximations are then based on the latest update.

Dead-reckoning uses a simplified model of the object's behaviour. Typical key state variables used in this model are position and velocity which may be linearly extrapolated to provide a low fidelity approximation. Higher fidelity may be achieved by incorporating other variables, such as linear acceleration and angular velocity, which are often needed by objects with highly dynamic behaviour, e.g. aircraft (Harvey et al., 1991; Le Saché and de Medeuil, 1993; McCarty et al., 1994).

This technique is very effective in reducing the amount of bandwidth required but the object behaviour produced in the ghost object can be sufficiently different from the normal to attract attention. This may, of course, be improved by increasing the complexity of the approximation, but there is a need to strike a careful balance between full and approximate simulation.

2.2.4.4 Total Replication

Simulating each object on each node may be required if the simulation is running over a very large distance (far/loosely-coupled). Receiving periodic updates from other processes when using partial replication is not practical when bandwidth is at a premium. Instead, only information that changes the behaviour of the mirrored objects is sent, thus permitting all calculations to be performed locally. Behaviour therefore appears correct everywhere (although maybe not at exactly the same moment in time) but at the cost of duplicated calculations.

2.2.4.5 Behaviour

What constitutes object "behaviour" and what form this takes is currently a topic of debate. In the strict object-oriented sense the data are the attributes, and the methods manipulate the attributes in a pre-defined way, e.g. modifying position over time. Therefore combining data and methods gives us the impression of behaviour.
However, the computational load required to support this object behaviour can be quite high, e.g. flight dynamics for an aircraft.

It is possible to classify object behaviour as either deterministic or non-deterministic. In general, objects that do not sample input devices are deterministic, whilst those objects that do, including those under the control of humans, are non-deterministic. For example, the decisions made by a robot car can be determined in advance whereas the behaviour of a virtual car being driven by a human in a driving simulator cannot be predicted (Hawkes, 1993). The ability to predict behaviour means that it is possible to overcome communication and system latency.

Roehl (1995) has suggested a refined classification scheme whereby deterministic behaviour is split into two sub-categories: static and animated. Similarly, non-deterministic behaviour can be Newtonian or intelligent. The state of a static object is constant and therefore 100% predictable for any given time; an animated object changes state over time but this is still predictable. A Newtonian object interacts with its environment but does so in a straightforward manner, whilst an intelligent object can have a complex behaviour and may be as unpredictable as a human.

In a similar manner, Roehl presents 4 levels of behaviour which may be used to classify the type of distribution used:

0. Direct modification of an object's attributes (static).
1. Change in an object's attributes over time (animated).
2. Series of calls to level 1 behaviours to achieve a task (Newtonian).
3. Top-level decision making (intelligent).

The most basic form of behaviour distribution takes place between levels 0 and 1, when information such as position and orientation are transmitted at every simulation update. Dead-reckoning falls between level 1 and 2. Attempting to distribute behaviour any higher is problematic unless the state of the simulation at each node is guaranteed to be exactly the same at any given time. Indeed, levels 2 and 3 may not even be implemented in software, they could be provided by human interaction.
An example of a level 2 behaviour system is the *Two-Point Paradigm* (Bryson, 1991). It is based on interaction in classical physics which may be taken as due to the forces that act pair-wise between physical objects. While many forces may act on objects simultaneously, the net action of these forces may be represented as the sum of the individual forces on that object from the other objects. To keep track of all these interactions an *Interaction Matrix* is used whereby each row and column represents an object and the entries are lists of interactions between the objects for that row and column. For example, Figure 2.1 shows the simple case of a bouncing ball. The ball is acted on by the floor in two ways: gravity pulling it down and bouncing which reverses the z component of the velocity. The floor is not acted upon by the ball. The ball's cross-reference entry (bottom right) updates its velocity from its acceleration and its position from its velocity.

This technique can be extended to include other types of interaction including those with the user. With regards to distribution, it is only necessary to send changes in the interactions between objects and details of any new objects from one node to another. Each node can then calculate the evolution of the VE on its own, which reduces the network bandwidth required per object.

**2.2.5 VE Modeling**

The issue of modeling the VE will be fully discussed in the next chapter, however there are two aspects which can be usefully addressed beforehand. Firstly, whether the system can support more than one VE simultaneously and/or how multiple VEs
are structured. Secondly, if any special provisions are made for users or participants in the VE.

2.2.5.1 Multiple VEs

Support for multiple VEs means that the system is effectively running parallel simulations using the same or different VE model. By using the same VE it could be possible to maximise the use of specific objects or areas of the VE (Roehl, 1995). For example, a virtual town hall could be used for meetings by different groups of people simultaneously.

If multiple, different VEs are supported then there is an opportunity to maximise the system's resources. Such an ability does, however, raise extra problems regarding scheduling, load balancing, etc. If the concurrent execution of VEs is available then a decision must be made as to whether an object may move from one environment to another and, if so, how this should be achieved.

Another possible use for multiple VEs is in the modeling process, where some or all of the properties of one VE are used to help speed development of another. The nature of the relationship between environments is important, as is the structure formed. One possible organisational technique is that of object-oriented inheritance where the attributes of one environment are inherited and augmented/extended by another environment.

2.2.5.2 Users

Typically, either the user is treated as a separate object or they are an integral part of the system. Also of interest, is whether multiple users can be supported or if only one may be present in a VE at a time.

Representing the user as an object has the advantage that it implicitly means that multiple users are supported, provided that there are enough input and output devices available. Added flexibility is provided if devices are not integrated into the user code directly, but exist as objects in their own right. The price of this object-oriented
structure is, of course, performance - the extra communications overhead increases system latency.

Either the user's representation can be described in the same manner as every other object or some extra functionality is provided for just this purpose. The latter case is usually used when the user is integrated into the system or a part thereof.

2.2.6 Time Management

The relationship between simulation time and real clock time may be any function as long as it is constant. The simulation clock is used as the basis for synchronisation of the VE either explicitly or implicitly. Implicit progression is when simulation time is related to real clock time: as the system clock changes, so does simulation time. Explicit progression is change through notification from a remote source, e.g. a message timestamp or a special message that only occurs at the beginning of each time step.

An additional requirement in a distributed system when using implicit progression is to ensure that the real-time clocks on each node are synchronised. One possible option is the use of a Global Positioning System (GPS) receiver built into each node. A version of GPS was developed by the military - Precision Position System (PPS) - for keeping track of friendly forces. It works by sending a signal to 4 out of 21 active satellites which send back information from which both positional and time information may be extrapolated. A commercial version is available, Standard Positioning System (SPS), with reduced positional accuracy - 100m horizontally instead of 17m, etc. Time accuracy with PPS is 100 nanoseconds (ns) and 167 ns with SPS. Detailed information can be found in Dana (1995).

Alternatively, a software algorithm can be used such as the one presented by Le Saché and de Medeuil (1993) where a client requests the time from a central source. The client synchronises itself on the time at this central source via a couple of timestamped messages. The synchronisation formula is shown in Figure 2.2. After clock
synchronisation, delays can be measured as the difference between the send and receipt times of any given message.

\[ t = \frac{t_2 - t_0 + t_1}{2} \]

- \( t \): new time for client
- \( t_0 \): send time of request time message (client clock)
- \( t_1 \): send time of response message (server clock)
- \( t_2 \): reception time of response message (client clock)

**Figure 2.2 A Clock Synchronisation Formula**

The problem of clock synchronisation is also of interest to the Internet community. The Network Time Protocol (NTP) is an extension of the client/server approach such that it may be applied in very large networks world-wide. For an in-depth description of the protocol the reader should refer to Mills (1992). On the general subject of clock synchronisation, Mills notes that the accuracy achieved is directly dependent on the time taken to achieve it. In other words, a few measurements will suffice for accuracy with a second or so, whilst dozens of measurements over many hours will be required to achieve millisecond accuracy. The number and frequency of these measurements is, however, perceived to be relatively low and unobtrusive to normal network operations.

However, Liskov (1993) notes that clock synchronisation algorithms are based upon assumptions about clock rate and message delay. Clocks are, therefore, only synchronised with some probability, albeit very high. Subsequently, she also states that algorithms should preferably depend on clocks for performance and not for correctness.

### 2.2.7 Fault Tolerance

Kim (1995) describes a fault tolerant computer system as “… a system which can continue to operate reliably by producing acceptable outputs in spite of occasional occurrences of component failures, including those of both hardware and software components”. Fault-tolerance comes from reliability and availability (Milenkovic, 1992). System reliability can be provided by partial replication of important data and
duplication of key hardware, whilst the availability of the system is ensured by keeping multiple copies of the system's resources. Furthermore, a system may be deemed recoverable if it can revert to a previous state and robust if it is capable of surviving a hardware failure. However, one does not imply the other.

<table>
<thead>
<tr>
<th>Degree</th>
<th>Assumable Damages</th>
<th>Recovery Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>No loss of visible actions (i.e. output of actions or database update actions)</td>
<td>Action-level fault tolerance (recovery of an interrupted visible action)</td>
</tr>
<tr>
<td>3</td>
<td>Loss of one or more visible actions</td>
<td>Slow recovery of a service function (no loss of hardware)</td>
</tr>
<tr>
<td>2</td>
<td>Loss of one or more service functions</td>
<td>Partial recovery of hardware (service degradation)</td>
</tr>
<tr>
<td>1</td>
<td>Loss of all but a core set of critical service functions</td>
<td>Minimum recovery of core hardware (minimum critical services)</td>
</tr>
<tr>
<td>0</td>
<td>Loss of critical service</td>
<td>No fault tolerance</td>
</tr>
</tbody>
</table>

Table 2.2 Degrees of fault tolerance.

Five degrees of fault-tolerance have been proposed by Kim which are reproduced in Table 2.2. Degree-4 is the highest level of fault tolerance (reliability) and ensures that all actions are completed successfully regardless of fault occurrences. It is possible that recovery from a fault may take so long that there is no choice but to abandon execution of visible action(s), restore the system to a previous state and then start again. Degree-3 caters for this case whilst degree-2 provides service degradation when some less-critical components fail and cannot be recovered. In the worst case, only the minimum critical services can be recovered and maintained which gives us degree-1 fault tolerance. If even this last stand is not possible then it is not a fault tolerant system.

A common way of providing fault tolerance is redundancy which may be applied to both hardware and software components. For the purposes of this thesis we are primarily interested in software components. The availability of broadcast communications on networks has been a great boon to the implementation of redundancy. One solution is to have a node which eavesdrops all inter-node communications to keep an up-to-date copy of each node's state. This means that if a
node should fail (or a process on that node) then its state may be rebuilt quickly without replaying all the messages.

In distributed real-time systems it is common for the physical network to be duplicated, therefore providing a second physical communication path should the other fail. In such a system there is also a need for deterministic and reliable delivery or messages, which has provoked some researchers into investigating reliable multicast protocols (Grünsteidl and Kopetz, 1991).

2.2.8 Security

Current efforts in this area have typically been limited to the encryption of the data stream between nodes so that no unwanted party can listen in on the simulation. This could be done in software at the communications level or utilise special hardware. The complexity of the system protocol determines the degree to which security can be breached; a simple protocol may even permit unauthorised objects or people to participate in the simulation.

On another level, security also deals with access to sensitive information. Certain system services may need to be restricted, e.g. access to backing storage, or a group of objects may wish to share information with each other and no one else.

Obviously, such additions to the system architecture come at a price. Even data encryption hardware increases latency and a software-implemented access control system can eat away at CPU cycles.

2.2.9 Issues Summary

This section has presented a number of issues that must be addressed during the design process. Some of these are given higher priority than others and as such may not be accounted for in the final design. This is not because they are unimportant, merely because the field is new and the problems presented by the few issues addressed are quite significant. The next section looks at the current major system solutions.
2.3 Implementations

Ensuring a consistent and accurate environment must be the main goal of any human-in-the-loop simulator. Progressing the simulation at a constant rate, fast enough so that the participant may effectively interact with it, is the second goal. It is clear from the overview presented in the previous section that many of the design issues are entwined with each other. Deriving an architecture that correctly resolves each issue is a challenging task. This section examines some of the existing distributed VE systems and describes their overall structure.

2.3.1 SIMulation NETworking System (SIMNET)

SIMNET was the first system to prototype and demonstrate the feasibility of a distributed interactive simulation (Kanarick, 1991). It was initiated by the U.S. Defense Advanced Research Projects Agency (DARPA) and funded by the U.S. Army. This project involved many different companies but, to the author's knowledge, no academic institutions.

Some of the systems requirements were (Calvin et al., 1993):

- Capable of supporting 100s to 100,000s of entities.
- Entities are geographically distributed.
- Simulations are heterogeneous.
- Computations are distributed (no central site).
- Operates in real-time.
- Must be low cost.

In order to meet the last requirement SIMNET was based around an Ethernet network. The maximum bandwidth of Ethernet is 10 Mbps and was one factor in the failure to support the large numbers of entities originally specified. 250 of the original SIMNET simulators (nodes) are currently in operation throughout the world although a node is capable of simulating more than one entity. A good example of this was the provision for Semi-Automated Forces (SAFs) which are semi-autonomous objects that have a certain behaviour and are directed from time to time by a human operator.
To make the most of the available bandwidth and reduce the computational overhead of point-to-point links between nodes, messages are broadcast to all nodes regardless of whether they require the information or not. The SIMNET protocol is designed such that if a node should miss a message, it will temporarily hold out-of-date information which will be amended upon the next transmission.

A host processor for a SIMNET node is typically an embedded single-board microprocessor-based system, or a workstation. Usually Local Area Networks (LANs) are used to link nodes within a single site and geographically dispersed sites are linked using Wide Area Networks (WANs). Due to the real-time requirement, the WANs are either private lines or packet networks with gateways that provide real-time allocation abilities. For example, the Defense Simulation Internet (DSI), which spans the U.S.A, is a dedicated network that uses the TCP/IP protocol but is not considered part of the Internet (Locke, 1992). The need to dedicate a network to a simulation indicates the problems of geographically dispersed simulations.

2.3.2 Distributed Interactive Simulation (DIS)

The experiences with SIMNET led to DIS which, unlike SIMNET, is being developed as a standard for networked, interactive simulation by a committee. An important distinction between the two is that SIMNET is a working system, whereas DIS is only a protocol definition with associated guidelines and does not specify how the implementation should be structured. Even though its applications are subject to security, the standard is not; version 1.0 is now a published standard: IEEE 1278.
Version 2.0 of the standard (DIS, 1994) summarises the DIS concept as:

"... a time and space coherent synthetic representation of world environments designed for linking the interactive, free play activities of people in operational exercises. The synthetic environment is created through real-time exchange of data units between distributed, computationally autonomous simulation applications in the form of simulations, simulators, and instrumented equipment interconnected through standard computer communicative services. The computational simulation entities may be present in one location or may be distributed geographically."

2.3.2.1 Basic Architecture

The DIS architecture shows its heritage through its basic concepts:

- No central computer controls the entire simulation exercise.
- Autonomous simulation applications are responsible for maintaining the state of one or more simulation entities.
- A standard protocol is used for communicating "ground truth" data.
- Changes in the state of an entity are communicated by simulation applications.
- Perception of events or other entities is determined by the receiving application.
- Dead-reckoning algorithms are used to reduce communications processing.

When examining the communication services that DIS must provide (as dictated by the standards document), we find that data must be transferred between simulations in one operation, with or without first establishing a logical connection with the destination node. Data may be sent using broadcast, multicast or point-to-point and, on the issue of unreliable service, no acceptable limit is set on the amount of data that may be lost. As a comment on the performance requirements of the communications architecture we are told that it "...should provide a certain level of performance characterised in terms of throughput and delay. Both network delay and network delay variance should be minimised". Another document (DIWG, 1993) states that
the total network delay for tightly-coupled simulators, such as high-performance aircraft, should be less than 100 ms and less than 300 ms for other simulators, e.g. ground vehicles.

Each message, or Protocol Data Unit (PDU), has a 32 bit timestamp which specifies the time at which the contents of the PDU is valid as units of time past the current hour. This provides an accuracy of 1.676 microseconds and the timestamps used depend on whether system clocks are synchronised or not. If they are, then the timestamp is given in Universal Coordinated Time (UTC), if not, then the time is relative to the simulation application that issued the PDU.

Each PDU has an exercise identity field in the header which is an unsigned 8 bit number. A unique exercise identifier is assigned to each exercise occurring simultaneously on the same communications medium. In essence, DIS can support up to 255 (a value of 0 is not valid) parallel VEs.

2.3.2.2 Performance

The total number of entities that may be supported is not only a function of the communications medium but the error thresholds which are an integral part of the dead-reckoning algorithms. Katz (1994) provides us with a graph (Figure 2.3) showing how the number of entities a medium may support can be reduced by decreasing the threshold (and hence the computational load of the dead-reckoning algorithm) and increased by raising the threshold (which increases computational load). The results shown are part empirical data and part prediction based on those data.

A state-of-the-art DIS system is said to manage 8,000 entities on Ethernet (using a lax error threshold) and that the most expensive dead-reckoning algorithm in use consumes around 100 FLoating-point Operations Per Second (FLOPS) per remote entity. Interestingly, it is predicted that the original SIMNET goal of 100,000 entities will not even be reached using DIS over an FDDI (100 Mbps) network. In fact, the Close Combat Tactical Trainer (CCTT), which is being developed by the U.S. Army and Loral Federal Systems using DIS and FDDI, expects to ultimately handle only
851 entities plus audio communication traffic (Mastaglio and Callahan, 1995). All this assumes, of course, that the simulation node itself has enough computational power to simulate 100,000 entities.

<table>
<thead>
<tr>
<th>Bandwidth (Kbits/sec)</th>
<th>Number of Entities</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000 FDDI</td>
<td></td>
</tr>
<tr>
<td>10,000 Ethernet</td>
<td></td>
</tr>
<tr>
<td>144 ISDN</td>
<td></td>
</tr>
<tr>
<td>56 modems</td>
<td></td>
</tr>
<tr>
<td>9.6</td>
<td></td>
</tr>
</tbody>
</table>

![Graph showing DIS performance with different dead-reckoning accuracies.](image)

**Figure 2.3 DIS performance with different dead-reckoning accuracies.**

### 2.3.3 Naval Postgraduate School Networked Vehicle Simulator IV (NPSNET-IV)

NPSNET is a research project in the Computer Science Department of the Naval Postgraduate School. The project's goal is "... to promote the use, understanding and appreciation of VR" (NPSNET, 1995). NPSNET utilises SIMNET databases, both SIMNET and DIS networking protocols and has a number of key functional components:

- **Terrain database** defining the 3D surface, e.g. ground or sea, and the various features, e.g. roads.
- **Static models** such as buildings, trees, etc.
- **Dynamic models** such as vehicles, aircraft, etc.
• Display algorithms which perform geometrical and rendering calculations on the complete VE from a given viewpoint.

• Environmental effects which included smoke, clouds, waves, etc.

• Heads-Up Display (HUD), a 2D overlay which may be used for superimposing information on the 3D view of the VE.

• Networking component which supports both broadcast and multicast.

• Input options allowing the device(s) to be matched to the application.

Despite being DIS-compliant, NPSNET only implements a fraction of the DIS Protocol, namely the Entity State, Fire and Detonation PDUs.

There are a number of software components unique to the NPSNET implementation (Zyda et al., 1992b), notably the Physically Based Modeling package. The Physically Based Modeller (NPSOFF PBM) models rigid-body dynamics using a Newtonian framework (Zyda et al., 1992a). Properties may include linear and angular velocities, mass and centre of mass, elasticity and location and orientation information.

2.3.3.1 Improving DIS

Macedonia, et al. (1995) correctly note that SIMNET was constructed for small unit training and has passed on this heritage to DIS. For this reason simulations do not scale well and are not currently suitable for large scale VEs. A number of problems are outlined:

• Bandwidth and computational requirements.

• Multiplexing media.

• Managing static objects.

• Database replication.

It is predicted that a VE with 100,000 players (entities) would require 375 Mbps of network bandwidth to each computer participating in the simulation. Since each node needs to maintain the state of every entity in the simulation (albeit using dead-reckoning models), they will require an inordinate amount of processing power. "We conjecture that 1000 entities are the limit to which a single host can realistically
manage despite future advances in computer and graphics architectures.” These figures are in line with the performance graph in Figure 2.3 and also means that a more powerful network medium than FDDI will be required.

DIS goes to great lengths to prevent packet fragmentation by requiring that each packet is smaller than the maximum supported by the physical network. Unfortunately, this means that video and audio must be treated in the same way rather than in their more natural continuous forms. Support for these media at the transport or network layers, e.g. through the use of MBONE, relieves the application from the overheads of multiplexing and de-multiplexing.

The simulations usually contain large amounts of static objects, e.g. buildings, that must periodically send update messages even though their state has not changed, just in case somebody missed the last message. The entire simulation database must also be replicated at each node since there is no method of partitioning the database. These last two points show the expense of the DIS protocol, both in bandwidth and computational terms.

The reasons offered for these problems are four fold:

- **Event-State paradigm.** Since the simulation is stateless (a basic requirement for DIS) information has to be sent to every entity. This does not take into consideration the fact that the simulated systems “sense” the environment in different ways and therefore have different data requirements. Two geographically distant entities need not know what each other are doing until they are in much closer proximity to one another. By being stateless, the simulation is affected less by the unreliable transmission medium being used (broadcast).

- **Real-time trade-offs.** A real-time environment should avoid point-to-point communications between entities since this requires reliable communications such as the acknowledgement scheme used in TCP. Centralised databases
cause I/O contention, so the only course left is to use a connectionless method of communication such as UDP.

- **Middleware.** There is no software layer to mediate between the simulation and the network. DIS must use bridges for large scale simulations which are an order of magnitude slower to reconfigure than routers and the number of nodes is limited to tens of thousands. A network using routers is limited only by the address space.

- **Small scale origins.** SIMNET and DIS were only used, until recently, for simulating small scale environments. This shows in the choice of transmission protocol and monolithic construction suitable for distribution over a single LAN. Past simulations have been packed quite densely with respect to the size of the environment and this influenced the assumptions made about rates of activity and inevitably the DIS protocol itself.

Complete replication of the database is also grossly inefficient and some means of partitioning information is required. The proposed solution to this problem is an Area of Interest Manager (AOIM). The VE is split into a grid of hexagons - since they are regular in shape and have uniform orientation and adjacency. The division of entities amongst the hexagons is not strict and some entities may belong to more than one group at a time to avoid boundary and temporal aliasing. As the user moves through the VE, the groups behind them are paged out and more groups ahead of them are loaded in. The advantages of such a system include reducing the bandwidth needed to maintain the simulation, the localisation of reliability problems and the ability to make use of high speed networks such as ATM. ATM will probably support multicasting and its high bandwidth might permit the dynamic paging in and out of the hexagonal areas containing large amounts of simulation data.

### 2.3.3.2 VE Modeling

Since NPSNET is based on DIS there is little modeling infrastructure. The entities may be simulated in full any way the designer sees fit, the only requirement is that its
behaviour can be approximated through a dead-reckoning algorithm. All nodes connected to the same network simulate the same VE.

2.3.4 Minimal Reality (MR) Toolkit

This toolkit is aimed at supporting work involving user interface design and may be split into three layers: low-level device support, data processing and high-level services (Figure 2.4).

2.3.4.1 Basic Structure

The device drivers are provided as a client/server pair, the server directly interfaces with the device and the client provides library routines that communicate with the server. The second level massages the data received from the device drivers into a more usable format as well as providing data sharing services between workstations. Complex tasks that are often performed have been encapsulated in a set of high-level functions to form the last layer. These include system initialisation and data synchronisation. All communications on the same machine uses TCP.

One application runs on a machine at a time. Each application has a master process that initiates the execution of other programs in the application which are designated as either slaves or computation. There may be many slave programs which perform simple tasks such as rendering images. Computation processes perform compute intensive work and are usually located on a dedicated machine connected to the master machine via a network.

2.3.4.2 Packages

To aid interface design a number of packages are provided to handle some of the more complex functions. There are currently four packages: Workspace Mapping, Panel, Data Sharing and Peer, but the latter two are of most interest in this context.

The data sharing package provides a way of managing a data structure that may be shared between machines by periodically sending an update copy to the other
machines. The structure may be synchronous, in which case the receiving program controls its update, or asynchronous where the receiver does not have control (the default).

---

**Figure 2.4 MR Toolkit component structure.**

The peer package is a recent extension to MR Toolkit and provides the functionality to allow independent applications to communicate with each other via master processes (Shaw and Green, 1993). The slaves receive data from their peers via their master, i.e. slaves do not communicate directly with other slaves or computation processes. Application-specific information may be shared between machines using UDP to send messages to specific addresses. Each machine keeps a peer list which indicates their state, either active or inactive. A peer may become inactive deliberately, with the intention to join in later or not (as the case may be), or a peer may inadvertently become inactive. This happens when the local peer has not received any messages from the remote peer in the last 10 seconds. At this point the local peer attempts to re-establish communication. All peers are connected directly to one another which requires a lot of network traffic to maintain and, as a result, more than five networked machines is not recommended by MR Toolkit's authors.
2.3.4.3 VE Modeling

Platform independent object geometry and behaviour is described in a procedural programming language called Object Modeling Language (OML). An OML object contains code to generate the 3D geometry, controls how the object appears and code for implementing behaviour. The OML compiler produces an intermediate code that is executed by the OML interpreter which is embedded into the application program. An MR Toolkit program loads compiled OML descriptions, initialises devices, coordinates between devices and the objects, and calls the interpreter every graphical update. Therefore a program has to be written for every VE built.

To save time, a generic VE application has recently been added to the suite of programs in the form of the Environment Manager (EM). The EM is responsible for initialising the VE (using a script file), running both single user and multi-user VEs, and also provides facilities for monitoring the execution of the VE (Wang et al., 1995). Each user in a multi-user VE runs an EM which handles calls to OML code. The distribution of the VE is transparent to the OML objects which just see one unified VE. The objects may be classed as local - managed by one EM only - or shared in which case other EMs may load them. To reduce bandwidth, only those shared variables that have changed state are transmitted and the EM also supports dead-reckoning by sending OML approximation functions to the other nodes. Unusually, it is possible to disconnect from the shared environment, perform some work and then reconnect at a later date.

The user is an integral part of MR Toolkit, in fact the whole system is built around the user. It is possible for multiple users to interact within the same environment when machines are connected using the peer package. Only one VE is simulated at a time.

2.3.4.4 Data and Computation Distribution

Two forms of concurrency control are supported through the use of ownership and access permissions; the choice of scheme is left up to the designer. A shared state variable may be owned by only one EM at a time and that ownership may, if needed,
be transferred from one EM to another at run-time. The solution to the case where
the transfer message is lost during transmission (possible when using UDP) is left up
to the programmer to resolve. A shared variable also has one of two possible access
permissions: writable and readable. If the variable is writable then EMs other than
the owner, may write to that variable. If it is readable then they may only hold a copy
of its value and its owner will send out updates when necessary.

Each EM has its own copy of the entire simulation including the shared variables.
The identity of the owner is broadcast to every EM whenever ownership changes.
When a remote EM wants to make a change, it requests ownership of the variable and
then makes the change. In other words, each machine in the network that has a user
wanting to interact in the simulation takes it in turns to run the simulation, whilst the
others get the results and use dead-reckoning.

OML descriptions may be created and manipulated using the Jiandong Liang
Computer Aided Design (JDCAD+) which uses a hierarchical modeling system and a
6 degree of freedom (d.o.f.) input device.

2.3.5 Distributed Interactive Virtual Environment (DIVE)

DIVE was developed at the Distributed Systems Laboratory, Swedish Institute of
Computer Science (SICS) to aid their research into the distribution, collaboration,
interaction and multi-user aspects of virtual reality (Carlsson and Hagsand, 1993).

2.3.5.1 Distribution

The distribution model used in DIVE v2.2 can be conceptualised as a memory that is
shared over a network. An old version of the ISIS Distribution Package (v2.1) is
used to provide a mechanism for data sharing between systems (Birman et al., 1987).
Version 3.0 of DIVE was in beta-testing at the time of writing and no longer uses
ISIS\textsuperscript{4} which has been substituted for the SICS Distribution Package (SID2 - Hagsand, 1992) that provides similar functionality.

The database, which is completely held in memory, is partitioned into worlds. Worlds are implemented as ISIS process groups where each process actively manages its own replica of the database. A DIVE process can only be a member of one world at a time although it may travel between worlds. Each process consists of lightweight threads which are allocated a specific task, e.g. rendering, input/output management or updating the database. The consistency of the shared database is maintained by using mutually exclusive locks, multicast transmissions within the process group and distributed object locks. DIVE supports heterogeneous distribution and machines that are not equipped with graphics hardware can still run non-rendering components of an application.

2.3.5.2 Applications

Applications may be created using the provided C libraries and then run on one or more systems communicating over an Ethernet link using TCP/IP. Multiple applications (implemented as a process) may run simultaneously, modifying the state of the world database. The visualizer is a special application that uses selected input/output devices and enables the user to interact with the VE.

Objects in DIVE are allocated a globally unique identifier, a name and a position in 3D space amongst other information (Andersson et al., 1995). They may also have one or more graphical representations. Composite objects are formed by grouping objects together hierarchically. Objects are stored locally in main memory, e.g. during creation, and may be shared over the network using a replication mechanism, i.e. after creation. Object information specific to an application is maintained by the application itself and is not distributed to other processes.

\footnote{ISIS is now a commercial package and is no longer free to academic institutions.}
All DIVE processes communicate with messages which may change an object’s state, a process’ state, or inform the recipient of a specific event. Applications may register call-backs for these events which may be used to indicate errors or user interaction.

Behaviour in DIVE is implemented as a state machine with, each arc referring to a particular signal type. A signal may be generated when a collision is detected, some form of user interaction has occurred, on some input, or when an application wishes to trigger a behaviour directly. A random signal is also available so that some form of random behaviour can be simulated. Current supported behaviours are limited to manipulating the object’s visual properties, spatial translation/orientation changes, generating a sound or triggering a behaviour in another object.

2.3.5.3 Users

Each user has their own personalised body-icon which is used to represent them in the world. The icon may be made of many parts, e.g. head, eyes, ears, hands and a visor. Each of these components serves a purpose. For example, each eye specifies a viewpoint from which the graphics display is generated and any object manipulated by the user is usually attached to one of the hands.

Vehicles provide a translation between data from input devices to actions in the VE. Several simple vehicles are provided with the system such as a mouse vehicle and one for monitoring head and hand movement when using an HMD. New vehicles may be created using the DIVE Application Programmer's Interface (API).

I/O handling and user representation is therefore integrated into the user object. Multiple users are supported as are multiple worlds which may be entered through gateways. Since each world possesses the same properties, there is no problem with object migration.

2.3.5.4 Time

Clocks in DIVE are not synchronised apart from system-level synchronisation using NTP and it is assumed that clock rates are equal on all machines.
2.3.6 Distributed Virtual Environment System (dVS)

Division build their own parallel processing computers which are currently based around INMOS Transputers, Intel i860 microprocessors and a number of ASICs. Their goal is to provide a seamless software environment to the VE designer which has resulted in the development of dVS (Grimsdale, 1993). Since its conception, dVS has been ported to Silicon Graphics, Inc. (SGI), Hewlett Packard and IBM workstations.

dVS v2.0.4 augments existing operating systems to try and provide the best possible performance over these platforms. It is organised into processes that perform certain tasks called *Actors*. There are actors for generating visuals, producing audio, performing collision detection, monitoring 6D trackers and many other tasks (Division, 1994). The user's application is also built from user supplied actors.

![dVS system architecture](image)

**Figure 2.5** dVS system architecture.

The essential components of dVS are shown in Figure 2.5. At the core is a distributed database (VL) which may be accessed by actors through the VL Library. The VC Toolkit provides higher-level functionality for the manipulation of objects and makes calls to the VL Library to achieve this task. The *Agent* is a special actor which handles updates to the local database and informs remote systems of the changes. One agent assumes the role of the *Director* and is responsible for coordinating all database updates. Communications between agents are performed using the Division Session Network (dSN) software layer.

45
2.3.6.1 Database Structure

An object class in dVS is called an **Element**. An **Instance** of an element may be created and is the unit of communication between actors. Before elements can be defined and instanced, an **Environment** must be created. A root environment is always created by default when an environment database is created (owned by the Director) and subsequent environments may be arranged hierarchically. New environments may be created by any actor at any agent. **Containers** can be defined which consist of one or more elements and are treated as an atomic quantity. A new element definition is written using C-like syntax and passed through a pre-processor which produces the relevant VL data structures and library routines as C source code. These source code files are compiled and linked into the application executable.

2.3.6.2 Database Synchronisation

Actors **hold** an element and by **extracting** that element an actor may change the state and then commit it using an **update**. Any actor holding the element will be informed of the change in state through an **event**. An actor can register interest in (hold) either elements or instances, an action that is environment specific, i.e. updates to sub-environments are not reported. This process is complicated if the item of interest is part of a container. There are actually 3 cases that must be catered for:

1. Interested in a container and a sub-element changes => the whole container is reported as having changed.

2. Interested in a sub-element and the container changes => the sub-element is also reported as having changed.

3. Interested in a sub-element which is subsequently changed => report a sub-element change.

Application tasks have no direct access to VL to avoid contention when two applications try and access the same information. All data accesses to the databases are therefore made by copying. dVS provides a choice of three different synchronisation methods to help maintain database integrity.
1. None. Updates are sent asynchronously and any duplicate events detected before the event reaches its destination are folded into one, i.e. only the most recent update will be processed.

2. Local. Locks the event and associated data until all destination actors within the domain of the local environment database have processed the information. This event is also propagated to remote databases if required.

3. Global. Similar to a local synchronisation event except the lock is performed across all remote databases and as such can be time consuming when acquiring the resources.

Synchronous updates are not supported by VL. These are viewed as expensive, used in only a few special circumstances (although no examples are given) and not the way to maximise performance (section 2.3.6.4).

The agent monitors changes to the local database and distributes these changes to other agents on other machines if interest in those items has been previously registered. Only knowledge about other agents and their current interests is maintained by any given agent, which means updates are sent direct to the relevant agents thus avoiding the need for broadcast. Since only objects that are being held are propagated to remote databases, it is possible for one such object to reference another which does not exist locally. It is up to the application to ensure that it has registered interest in all necessary objects. Agents are allocated a port number which is held in a configuration file, allowing physical machines to connect or disconnect at run-time.

2.3.6.3 VE Modeling

The VC Toolkit supports a number of specialised elements which it calls Virtual Objects. The basic element is VCObject which may be decomposed further into other VCObjects and so on. The other standard elements which are held within a VCObject are VCAudio, VCBoundary, VCConstraints, VCLight and VCVisual. Each of these describes a certain number of logically related attributes and are often associated with a particular actor, e.g. VCAudio elements are monitored by the VSOUND actor.
The collision detection actor monitors VCBoundary elements and notifies the two relevant parties when a collision has occurred. Whereas the VIZ (visualisation) actor is interested in VCOBJECT, VCLIGHT and VCVISUAL elements.

Users are represented by a Body actor and therefore there may be multiple users in the same environment. The body actor is also abstracted away from the necessary I/O devices which exist as separate processes and can be assigned a special representation. It is unclear whether an actor from one environment can move into another.

2.3.6.4 Synchronisation

When a network of machines starts up, the first node to complete initialisation sets the time on the other machines to its own. No time synchronisation is performed thereafter. All messages are timestamped but this information is used to discard tardy messages that have already been superseded. dVS never waits for the arrival of a specific message and thus there is no lock-step synchronisation between nodes.

2.3.7 Waterloo Virtual Environment System (WAVES)

WAVES was formerly known as Highly Interactive Distributed Real-Time Architecture (HIDRA) and is targeted at low-cost platforms that use low-bandwidth media for communications, e.g. telephone lines (Kazman, 1993c).

2.3.7.1 Basic Architecture

The basic components of the WAVES architecture are shown in Figure 2.6. Each Host simulates a subset of objects and provides certain services to each object, e.g. collision detection, rendering, etc. Whilst cyclically updating their set of objects, hosts periodically broadcast the state of their local objects to other hosts. Major I/O events, e.g. user input, are communicated each cycle to maximise fidelity. The communications between these hosts are done over virtual connections, mediated by a number of Message Managers. Connections may also be filtered so only messages of interest are sent to the hosts. The message managers are also given the ability to delegate direct point-to-point links between hosts in special circumstances, e.g. a line
carrying a video signal. Under WAVES, a VE may be distributed over a network of message managers, with the allocation of hosts to each manager being determined by a dynamic clustering algorithm. Objects have explicit behaviour models which aid load balancing, support dead-reckoning and may be used to predict an object’s state in order to combat latency. The ghost objects that reside on a host are called *clones* in WAVES. As with other dead-reckoning systems, some fault tolerance is provided in that if one host should go down, then the others can carry on using their current behaviour models.

![Diagram of WAVES architecture](image)

**Figure 2.6 Basic WAVES architecture.**

Load balancing is performed on each host based on several criteria: the host’s processing power, the number of objects on the host and how closely related the objects are (Kazman, 1993b). The host sends its current load and their maximum possible load to their local message manager. When the host detects its load has risen above its maximum, it sends another message to the message manager indicating which object it would like to get rid of. Another host is found for the object or, if no suitable host can be found, the transfer request is refused.

Users should be representable as objects providing there are sufficient input/output devices on a host and this would also imply that multiple users can be supported. It is unclear whether it is possible to execute multiple environments.
2.3.7.2 Distribution of Responsibility

To solve the problem of area management, WAVES uses a special Area Manager which is paired with a message manager (Kazman, 1993d). The area manager maintains a list of viewable areas for a given viewpoint, one per host. When the list changes, the message manager's filtering criteria for a given host is changed so that only those objects in the host's viewable areas are sent to it. Since the area manager only changes message filters, it can be added or removed from a WAVES system without disturbing anything else in the system. To overcome rapid changes in area, WAVES proposes to use an object's behaviour model to anticipate the changes and send filter requests in advance. To avoid the problem of all users occupying a small number of areas and causing a bottleneck, there may be many managers in the system and they may balance their loads dynamically.

Interactions between objects are specified externally in interaction detection and resolution (IDR) agents (Kazman, 1993a). The world view maintainer contains the description (world attributes) of the environment that the objects operate in; a view controller which dynamically manages the inventory of agents which may be interfacing, and an inventory of all the objects which exist in the world (world objects). If IDR takes too long then the world may be broken into a number of areas, each with their own IDR facility. Each IDR server contains a production system, which allows the system designer to create arbitrary constraints on an object's state in the form of rules that are evaluated each execution cycle. Each IDR server contains a "theatre map" that plots the locations of all objects in the theatre and raises an exception when two objects attempt to occupy the same space. IDR servers can be designed to handle particular types of interactions: spatial, temporal or semantic. IDR can therefore be used to detect interactions within a spatial threshold as a sort of prediction mechanism to accommodate lags in the system.

In summary, object behaviour is defined within the objects, interactions are defined within IDR servers and the environment is defined within the world view.
2.3.8 AVIARY

In the AVIARY model a distinction is made between objects that are presented to the user through different media: Demons are the pieces of software that implement an object and Artifacts are the manifestation of the demon in the VE (Snowdon et al., 1993; Snowdon and West, 1994; Snowdon, 1995).

2.3.8.1 Basic Architecture

A virtual world is seen as a container for artifacts and a set of constraints on those artifacts and behaviour. The sole World Object represents a virtual world, acting as a container for artifacts, storing the identities of demons, details on the objects providing other services, and information shared by all objects. The actual artifact definitions are not held within the world object, but since the artifacts may be accessed through it, this information can be obtained indirectly.

The Environment Database (EDB) provides a spatial management service to other objects. When a demon moves, it sends a message to the EDB, which prompts a collision check for that object. The EDB then informs the relevant objects of the collision and they may then react as they please. To prevent the EDB becoming a bottleneck, it may be split into one or more new EDBs which share the existing workload (although this has not been implemented yet). In addition, separate EDBs may be employed for different media and therefore operate in parallel.

Object Servers provide an execution environment for demons, handling object creation/destruction, messages from other objects, memory management and scheduling. Inter Process Communication (IPC) between all types of objects is supported without restriction. One object server is allocated to each processor. Security-wise, each object controls access to its own data and may therefore protect any sensitive information.
Figure 2.7 AVIARY component schematic.

Only one Virtual Environment Manager (VEM) is present in the whole system and provides services to ensure that the integrity of the VE is always maintained. This includes the assignment of identifiers to objects (aiding dynamic object creation) and also to classes and messages. This last mechanism ensures that objects that understand the same messages but have been implemented differently can still communicate with each other. Complementing the VEM is the World Manager which maintains a list of all the available services provided by objects. This enables any object to look for another object providing a service that it requires, e.g. visual rendering or collision detection.

The issue of time synchronisation is resolved in AVIARY by making use of real-time clocks on each node. Simulation or world-time can, however, be scaled relative to real clock time.

Both synchronous and asynchronous message passing is supported with both multicast and point-to-point links used to transfer the message. To prevent deadlock, the object server is multi-threaded so that it is always ready to respond to an external event.
2.3.8.2 VE Modeling

Behaviour of the artifacts is dictated by the methods defined for the creating demon which actually consists of two parts: artifact-specific and world-specific. All features that are shared by all objects in the world are held in the world-specific part and those unique to each class of demon in the artifact-specific part. This separation of attributes aids migration from one world to another. Demons can make use of services provided by any other kind of object and can inherit classes or define existing classes to extend its capabilities.

Multiple worlds are an important part of AVIARY's design since each may require a different interaction metaphor and its own laws and properties. The user is permitted to travel between worlds by using Portal objects that may appear as artifacts in each virtual world. When a demon moves between worlds the world-specific part of the demon is replaced by that of the new world whilst the artifact-specific part remains unchanged.

A demon may represent an application or a user, either way it is likely to need access to input and/or output devices. Input objects control input devices, sending data to all interested objects only when there is something new to send. Output objects monitor a particular location in the world and display a representation in the chosen media.

Users are represented by demons and are decoupled from the system and I/O devices. Although there does seem to be provision to integrate I/O into the user demon if performance dictates.

The current implementation is written in C with object-oriented features, including multiple inheritance added through macros. These macros create an internal data structure of class descriptions, object instances, etc. This data may be communicated to another machine thus supporting object migration, although no load balancing checks are currently made to see whether this is required.
2.4 Summary

All of the systems examined here are trying to achieve interactivity, but none are real-time in the traditional sense and therefore do not support real-time displays. These are important aspects of a VE system and the impact of supporting them is discussed in sections 3.3 and 4.3. This classification has therefore been left out of the feature summary table (Table 2.3).

2.4.1 Communication Organisations

Typically there are \( n \) processes in a VE which need to communicate with each other. Using a point-to-point communications system, a link must be established between every process or a central server established, forming a hub. In the former case this will require \( n(n - 1) \) links and in the latter, \( n \) links, although total centralisation can place a burden on the central server which can quickly become a performance bottleneck. Conversely, administrative processes often need to monitor most (or all) transmissions and make according actions, e.g. dVS’s Director and the Message Managers in WAVES.

Whilst broadcast relieves the overhead of maintaining links, it floods the network with messages which are either an inconvenience (on shared networks), or wasteful (on dedicated networks) because in large VEs not every process needs to know what all the others are doing. Area management can be used to determine who needs to know what, but cannot use broadcast as the transport mechanism. Maintaining a number of point-to-point links is one solution but with the complications already outlined above. Multicast provides a way of overcoming these disadvantages (as demonstrated by the AOIM in NPSNET) whilst still retaining the low transmission overhead, but it is not widely available and is, like broadcast, unreliable.
<table>
<thead>
<tr>
<th>Feature</th>
<th>DIS/SIMNET</th>
<th>NPSNET</th>
<th>MR Toolkit</th>
<th>DIVE</th>
<th>dVS</th>
<th>WAVES</th>
<th>AVIARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communications</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transport Mechanism(s)</td>
<td>Point-to-Point, Broadcast, or Multicast</td>
<td>Multicast</td>
<td>Point-to-Point within node and Broadcast between nodes</td>
<td>Multicast</td>
<td>Point-to-Point</td>
<td>Point-to-Point</td>
<td>Point-to-Point and Multicast</td>
</tr>
<tr>
<td>Targeted Bandwidth Structure(s)</td>
<td>Unspecified</td>
<td>10 Mbps +</td>
<td>10 Mbps +</td>
<td>10 Mbps +</td>
<td>14 Kbps +</td>
<td>10 Mbps +</td>
<td></td>
</tr>
<tr>
<td>Data Management</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Organisation</td>
<td>Total Replication</td>
<td>Total Replication</td>
<td>Active &amp; Passive Partial Replication</td>
<td>Total Replication</td>
<td>Passive Partial Replication</td>
<td>Complete Distribution</td>
<td>Complete Distribution</td>
</tr>
<tr>
<td>Localisation Support?</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Computation Management</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Organisation</td>
<td>Partial Replication</td>
<td>Partial Replication</td>
<td>Total/Partial Replication</td>
<td>Total Replication</td>
<td>Partial Distribution</td>
<td>Partial &amp; Complete Distribution</td>
<td>Complete Distribution</td>
</tr>
<tr>
<td>Behaviour Level</td>
<td>1</td>
<td>1</td>
<td>0/1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>VE Modeling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Environment Management</td>
<td>Parallel</td>
<td>Parallel</td>
<td>Single</td>
<td>Multiple</td>
<td>Parallel</td>
<td>Unknown</td>
<td>Multiple</td>
</tr>
<tr>
<td>User Support</td>
<td>Multiple</td>
<td>Multiple</td>
<td>Multiple, Integrated (possibly)</td>
<td>Multiple, Integrated, with Representation</td>
<td>Multiple, Decoupled with Representation</td>
<td>Multiple, Decoupled (probably)</td>
<td>Multiple, Decoupled</td>
</tr>
<tr>
<td>Time Management</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Progression Method</td>
<td>Implicit or Explicit</td>
<td>Implicit or Explicit</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>Unknown</td>
<td>Implicit</td>
</tr>
<tr>
<td>Node Synchronisation</td>
<td>UTC or None</td>
<td>UTC or None</td>
<td>Programmer</td>
<td>None</td>
<td>None</td>
<td>Unknown</td>
<td>None</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>Degree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>3/0</td>
<td>0</td>
<td>0</td>
<td>3/0</td>
<td>0</td>
</tr>
<tr>
<td>Security Method(s) Employed</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 2.3 Distributed VE system feature classification summary.
2.4.2 Transport Mechanisms

Deciding whether to use a reliable message delivery service or not is a key decision in the design of a distributed VE system. NPSNET, MR Toolkit and other DIS-based systems use UDP between machines. DIVE uses multicast exclusively and AVIARY uses it for messages that need to be sent to many processes, but this is under the control of the application programmer. The only two systems that use a reliable service exclusively are dVS and WAVES. Both make use of a known network configuration and thus known addresses, to distribute the messages. If an unreliable service is used then the software protocols must reflect this decision and a degree of fault tolerance provided.

Even with the implementation of these two steps, loss of messages (or their delayed reception) will, inevitably, have an affect on the user interface. The effect could be anything from a slight glitch or jump in the display, to temporary loss of service. If these counter-measures are not taken then the designer is relying on a large number of variables holding true to keep things running, e.g. plenty of bandwidth available, network interfaces fast enough to capture packets, etc. Of those systems reviewed that use multicast/broadcast, NPSNET and MR Toolkit account for lost messages. Both use exported behavioural models but MR Toolkit actively encourages a machine to disconnect and reconnect during a simulation by providing appropriate API functionality. To the author's knowledge, DIVE and AVIARY do not make any provisions for lost messages, the consequences of which are unknown for both systems.

2.4.3 Bandwidth Implications

All of the systems use existing networking technology so it is unsurprising that most are currently implemented using Ethernet. The DIS standard does not actually specify a bandwidth but the author does not know of any implementation using anything less than 10 Mbps. WAVES' target of 14 Kbps is laudable but there is precious little bandwidth to play with. Without compression, 14.4 Kbps will support a data rate of
approximately 1.31 Kbytes/second. The compression supported by modern modems could improve on this if there were repeating patterns in the data stream, like those found in ASCII text. However, the likelihood is that the messages sent between nodes will contain extensive binary data and thus compression will do little good. This figure does not, of course, include transport protocol overheads which may reduce the data transfer rate substantially (section 4.2.3).

The fact that the available bandwidth for a given process will vary during execution is a compounding factor. This affects both reliable and unreliable services and, depending on the criticality of the system, can at the very least wreak havoc on system performance. The author believes that the ability to allocate channels of fixed bandwidth for a fixed period (as supported by ATM), is essential to the development of distributed VE systems. Only then will communications become deterministic and thus release the designer to concentrate on other issues.

2.4.4 Distribution & Scaleability

Communications latency affects all systems, regardless of architecture, however, it is the largest enemy of scaleability. As the distance between nodes increases so will the latency and unless the system protocol and structure is modified to account for this, performance will degrade beyond acceptable levels.

Each of the current systems reviewed address one of Kleinrock’s classes with a possibility of application in another if the conditions are right. None attempt to address more than two and certainly no changes are made to the system architecture to help it adapt. Each form of data and computation distribution has advantages and disadvantages. All can be applied successfully in a near/tightly-coupled system but as we move through far/tightly-coupled into the far/loosely-coupled classification, so the solution weaknesses become more apparent.

\[ \frac{14.4}{11} = 1.309 \text{ Kbytes/second} \] (assuming 8 data bits, 1 start bit, 2 stop bits and no parity).

5 14.4/ 11 = 1.309 Kbytes/second (assuming 8 data bits, 1 start bit, 2 stop bits and no parity).
Complete distribution of both data and computation is a victim of increased latency since all accesses and data modifications have to be communicated to their source. The worst-case task would be the monitoring of a piece of information, performing an action when it reaches a certain value and then modifying it. This would require a message to get the latest value and possibly another to modify it every simulation step. If this task was performed on many objects it could saturate the network. Both AVIARY and WAVES use this approach. Active partial replication, as used partly by MR Toolkit, also has the same problem. Whereas data would be accessed via an object interface with complete distribution, copies of whole chunks of object state can be distributed with active replication.

Partial replication of data provides slight relief from this symptom by supplying a mechanism that will send any interested party a copy of the relevant portion of state (or behavioural model) only when it changes. Not only does this reduce bandwidth consumption, but also the computational load because the task function is only executed when an update is received, rather than at every simulation step. Modifications can be made by sending the instruction to change data back to the source. A slight variation on this is partial computational distribution where changes are made locally and communicated back to the owner, or, as in dVS, committed to the shared database. If the latter method is used, locks must be used to preserve data integrity. Lock acquisition and release can lead to deadlock (section 2.2.3.3) and are inherently undeterministic and thus unsuitable for a real-time system.

Complete data distribution has the advantage that data is only stored in one place, while partial data replication duplicates parts of the environment’s state, thus consuming more resources (DIS/NPSNET). This pales into insignificance against total replication where the complete environment state is duplicated. In a high bandwidth configuration this is a waste of resources, but it is the only solution when the distance between nodes is large and latency is high. The largest challenge in this case is to keep the replicated databases in synchrony. Transmitting modified segments of environmental state between databases is not a viable option. Partial computational replication would seem to be a possible solution.
The usefulness of exporting behavioural models can be shown clearly by once again considering the goal of distributed VEs over a 14.4 Kbps telephone line. A level 0 behaviour system would likely send a position and orientation update for each simulation time step. Assuming 6 x 32 bit floating-point numbers (3 for position and 3 for orientation) plus, say, another 16 bits for an object identifier gives a total of 208 bits or 26 bytes. Using our previously calculated data rate of 1340 bytes/second we can determine that 1340/26 = ~51.5 messages that can be sent per second. Assuming a modest 15 Hz update rate, this permits us to send updates to ~3.4 objects. If more bandwidth is available initially then this is quite a tempting, easy solution and is used by DIVE, MR Toolkit (at its lowest level), AVIARY and, to a lesser extent, dVS. If a higher level behavioural model was supported, such as dead-reckoning, then messages would be sent at a much lower rate (depending on the object's behaviour) thus permitting more objects to be supported.

However, level 1 behaviours still require messages to be sent quite often and it would be quite easy for the databases to get out of synchrony considering the latency. Instead of informing each other of deviations from the predicted behaviour, it would be more sensible to totally replicate the computation and only inform each other of changes in object behaviour. This could be an update of the behavioural description effected by software, e.g. level 2 behaviour, or by a user, e.g. level 3. Bryson's two-point paradigm (2.2.4.5) is representative of the kind of information that could be sent.

Load balancing and process migration are best applied in a tightly-coupled system. There is obvious application for these techniques when using complete computational distribution and they can also be applied to systems using partial replication. With a large number of ghost processes and area management there are likely to be those that are accessed more frequently than others. Spreading the computational load evenly whilst minimising the distance between communicating objects could greatly improve performance.
2.4.5 Time

Most of the systems reviewed do not seem to have any policy on time management. AVIARY uses the implicit model for clock synchronisation which is less than full-proof. Clock oscillators can drift (as any network administrator will testify) and need to be constantly corrected. The most common method for doing this is NTP which is adequate for non-time-critical work where second accuracy will suffice. When dealing with multiple updates per second this clearly will not do. With extra effort over a longer period of time it is possible to synchronise clocks to millisecond accuracy using NTP, but the author feels that this may be inadequate when dealing with 33 ms time spans (for a 30 Hz update rate). Ideally, each node would be equipped with a Standard Positioning System which would ensure that all machines throughout the world were synchronised to within 167 ns. Unfortunately, the current cost of this technology would probably be prohibitive so solutions like NTP are the best remaining choice for systems using implicit time models. Indeed, if clock synchronisation is needed in MR Toolkit or DIVE, the designers have assumed that NTP would be used.

The explicit time model uses timestamps in messages for various purposes such as informing them of the send time, the time at which the message is valid, etc. DIS/NPSNET uses a timestamp format which can specify a time up to an hour after the current hour, to within an accuracy of 1.676 microseconds. However, there seems to be no suggested methodology of ensuring that each node has the correct current time. In this instance there would seem to be a requirement for both models to be used together to manage simulation time.

It might be possible to use explicit time progression exclusively within systems that use complete/partial computational distribution or partial replication, but when total replication is used a common reference is required.
2.4.6 Fault Tolerance

Those systems that export behavioural models (section 2.2.4.5) implicitly support a notion of reliability (degree-3). Failure to receive an updated model, because the source host is down, can be remedied when the host rejoins. DIS ensures this by requiring that no one machine controls the simulation. MR Toolkit permits a node to leave and rejoin the simulation but this does not really constitute robustness since leaving and rejoining relies on using the correct protocol. WAVES does export behavioural models, but there is no mention in the available documentation that states fault tolerance as a design goal.

None of the systems pursue the goal of availability through duplication of resources, probably because they are at a premium. Total replication of both data and computation is done by DIVE which would put it in the best position to provide fault tolerance, although this is not a stated goal. When the faulty node recovers, another node in the simulation can send it a complete copy of the current environment state. Recoverability is not supported by any of the systems and the only true robust systems are those based on SIMNET/DIS.

If interaction is a high priority then degree-4 fault tolerance is the most desirable and may even be considered as the only usable type. Any faults managed at a degree below this would be reflected as a disconcerting change in the VE display. This may manifest itself as anything from a small “jump” in continuity (degree-3) to a total loss of realism (degree-1).

Rather ironically, the least reliable transport mechanism - broadcast - is also the best way of providing fault tolerance: through redundancy. The incorporation of a special process/node in the network that listens into every message and maintains a state backup using point-to-point links would place an unacceptable overhead on communications. It would require two messages to be sent for every communication rather than just one. Fortunately the reliability issue is being dealt with (sections 2.2.2.3, 2.2.7) which will remedy one of the weaknesses of any system that uses broadcast techniques.
2.4.7 Security

This is an issue that none of the current systems fully address. This is not too surprising since all of these systems are used as tools for researching the field and security can get in the way. An encrypted data stream is not particularly helpful if you wish to monitor message passing, nor is access control when you are experimenting with object interaction metaphors. AVIARY makes a token gesture by putting each object in control of its own data. This is not an added feature, this ability comes with the adoption of an object-oriented structure. An object’s methods may be coded in such a way to vet access but AVIARY provides no built-in/automatic security layer.

2.4.8 Modeling

With the exception of MR Toolkit and WAVES, all of the systems support the concept of multiple VE's in one way or another. DIS supports multiple exercises which take place in the same environment, whether these exercises can interact is not clear. DIVE assigns a multicast group to each environment so messages are not processed unless the user is present in that environment. dVS can support different environments but there is no evidence to suggest that elements in one environment can move to another at run-time. All objects in AVIARY occupy one of the available VE's which are designed as a hierarchy of worlds, each one building on the properties of the parent. Objects may also migrate from one world to another, a feature shared by DIVE. However, in AVIARY worlds may possess different properties whereas DIVE worlds would need to be programmed identically to facilitate migration.

All the systems support multiple users in differing ways. MR Toolkit might support more than one user if each had their own workstation and was sharing the same database. The WAVES literature does not specifically state that it can support many users, but its general structure of hosts and I/O devices infers that it does. In DIVE and MR Toolkit, the user is an integral part of the system, in fact they are built around the user. DIS, dVS, WAVES and AVIARY do not distinguish a user from any other object except that it may have various I/O devices connected to it. All of these can be used to simulate VE's with no human participation whatsoever. Despite this
treatment, dVS does seem to emphasise the ability to specify a special user representation in the VE in a manner similar to DIVE. The latter, however, also uses this representation to configure the required I/O devices.

2.4.9 System Summaries

2.4.9.1 DIS-based Systems

DIS and SIMNET would have originally been classified as near/loosely-coupled but DIS is now trying to move on towards far/loosely-coupled. The problems with such a move have been discussed in this summary and in section 2.3.2. NPSNET is being used by the Naval Postgraduate School as a testing ground for new ideas and concepts to help DIS make this transition. Despite the DIS community's advocation of the protocol's applicability to non-military VEs, the author feels that it will always be of restricted use due to its constrictive definition. All messages sent between objects have to be defined in advance and of the dozens already defined only one of them is of general use: the Entity State PDU. The other PDUs deal with explosions, logistics support, etc., which are inherently military-application specific.

2.4.9.2 MR Toolkit

This system is used to aid research into user interfaces and is accordingly designed around the user. It does its task well but its lack of generality limits its applications in the same way as DIS-based systems.

2.4.9.3 DIVE

DIVE is more flexible than DIS and MR Toolkit, but its use of total replication and an unreliable message delivery system make scaleability a real issue.

2.4.9.4 AVIARY

Of all the systems reviewed, AVIARY is the most flexible but shares another problem with the others in that it will have problems scaling up to larger VEs. The use of
complete distribution has limits and must be supplemented with other forms of data/computation management, requiring changes in the system's architecture.

2.4.9.5 WAVES

There is only limited information available on this distributed model although the literature states that a prototype implementation is being developed. The inclusion of low bandwidth communications is cause for concern and catering for this could compromise the design.

2.4.9.6 dVS

A restriction shared by all of the systems presented here is the difficulty with which the VE definition is changed. dVS requires the basic components and structure of the environment to be scripted off-line, pre-processed, compiled and linked in with the Actors.

Its exclusive use of point-to-point links may also prove to be detrimental to performance when larger networks of dVS machines are attempted.

2.4.10 A New Architecture

From the analysis presented in this chapter, it is possible to extract those features that effectively resolve the presented issues and derive a new architecture for distributed VE systems. This is presented in chapter 4 following a closer look at a couple of aspects which deserve more attention: modeling and displaying VEs.
In this thesis, not only are we interested in the technical aspects of distributed VE systems - their architecture - but also in the methods used to model VEs. To better understand what we are trying to achieve when we model a VE, our natural environment is examined and ways of defining and classifying VEs are explored. To conclude this abstract examination of environments, a number of modeling processes that may be used to capture the essence of the environment being modeled are discussed.

As the reader knows, we interact with any environment via our senses. The information we gather from these senses is processed by the various perceptual systems in our brain. An effective VE system will generate displays that enable human perception, e.g. visual and auditory, to operate naturally. If the VE displays present the information in a confusing way, then the VE system is not doing the model of the environment justice.

To illustrate this point, the implications of misusing the visual display are discussed - currently the norm rather than the exception. Rectifying the problems with the way in which this display is used has ramifications for VE system design. These technical
details are discussed in this chapter as a prelude to the consideration of the more
general system requirements presented in the next chapter.

3.1 A New Modeling Paradigm

There are many questions that should be asked when designing a VE. What sort of
information should be provided? How should it be structured? How can it be
described? These questions face all VE designers, whether the environment is
intended for data visualisation, teleoperation or vehicle simulation. In the hope of
gaining a better understanding of the task at hand this section examines our natural
environment. How we interact with our environment, its important features and its
implications on VE design are discussed with the aid of several VE definitions and
classification schemes. By analysing how we interact with the real world we can gain
insight into how effective virtual worlds may be constructed.

For argument's sake, let us say that (for now) a VE is a synthetic version of our
natural environment. Logically, our next question would be "what is our
environment?" How do we describe the environment in which we live? This is a
question that has been given a great deal of thought by many people working in every
discipline: Physics, Psychology, Physiology, Philosophy, the Arts, just to name a few.
Each of these disciplines offers its own unique view on the subject. Regardless of
their definitions, which can be quite different, they are all valid and each has its own
place and use. Physics can provide us with information on how the environment is
constructed in physical terms of force, mass, energy, etc. Physiology deals with how
our body functions within the environment, Philosophy deals with more abstract
concepts, whilst Psychology concentrates on the more cerebral activities of our body,
including our perception of the environment that we are in.

3.1.1 Definition of a Model

Before we tackle the thorny issue of defining a VE, it is useful to give some thought
to what we mean by a "model".
A model is an implementation of a representation of an abstraction of a thing.

Barzel (1992), p27.

The thing being modeled is not part of the model, indeed the model is a simplification of the thing, consisting of a subset of the properties that make the thing. This subset is the abstraction and can be thought of as the set of ideas that underlie the model. As such, the abstraction is an entity without substance and therefore cannot be manipulated. The representation is a complete description of the model and is concrete in the sense that it may be edited, copied, analysed and contains sufficient information to build the model. It is possible to have many representations for any given abstraction and a representation to be shared by multiple abstractions. The execution of the model is the implementation, of which there may be many for any given representation, each with their own quirks.

This introduces us to a way of describing models referred to by Barzel as an Abstraction Representation Implementation (ARI) structure. Barzel presents ARI for use in physically-based modeling but it can be used to decompose most types of model. For example, the model of a computer program may be analysed using the ARI scheme: the conceptual specification is its abstraction, the design document is its representation and, naturally, the software itself is the implementation. If we adopt this methodology for the modeling of VEs then we must first find a suitable abstract model. From this we should be able to derive a suitable representation, maybe in the form of a language, and eventually the implementation of a system which can execute our VE.

3.1.2 An Ecological Approach

The nature of the environment and how it shapes the evolution of animals contained within, was a key concern of the eminent psychologist, James J. Gibson. When describing the Gibsonian approach, the key word is affordance (Gibson, 1979). The affordances of an environment are what it offers the animal in terms of action and
interaction, what it *provides* or *furnishes*, for good or bad, e.g. a fire can afford warmth but it also has the power to destroy. An important point is that affordances do not reside in the environment, they are the result of interactions between the animal and the environment.

Objects within the environment are classified as being either *attached* or *detached*. In Newtonian physics, all objects in space are detached, but from an alternative perspective it is obvious that some items are attached and cannot be moved without breakage. In order for an object to afford behaviour, it must be both detached and comparable in size to the animal under consideration. Exactly how small or large an object has to be until it does not afford behaviour is unclear, but those objects that are comparable can afford a wide variety of behaviours. Objects can all be said to have properties or qualities, e.g. colour, texture, composition, size, mass, etc. Orthodox psychology asserts that we perceive these objects insofar as we discriminate their properties or qualities, but Gibson suggests that what we perceive when we look at objects are their *affordances*, not their qualities. However, to perceive an affordance is not to classify an object, e.g. a stone is a missile but it can also be a paperweight, part of a wall, etc.

By describing the environment in terms of animals, Gibson rightly makes the point that each animal has its own view of the same environment. Or to put it another way: given an infinitely detailed environment, each animal will extract only that information which it needs. Because different subsets of the environmental properties are being used, the animal's *perception* of the environment (and the objects within it) will be different. For example, when we look at a tree we may be interested in it as a material for construction or maybe as shelter from the rain. A dog, on the other hand, may be assessing it for more basic needs.

There is a very simple reason for this situation. If you examine an environment in detail it will present properties that are conducive to certain animals and properties that make the environment hostile towards others. Look at any species that survives today and you will see an animal whose perceptual systems have evolved to
complement its environment. An animal implies an environment and an environment implies an animal.

If the affordances of a thing are perceived correctly, we say that it looks like what it is. However, when evaluating the properties of an object, it is important for us to take a step back and view them in the context of the environment and not just from the human perspective - a task that is easier to state than accomplish. Gibson's ecological framework has already motivated the design of a VE Computer Aided Design system (Smets et al., 1993, 1994). Familiar modeling tools such as hammers and saws are replicated in the VE and afford behaviours found in everyday life, although they are not limited to these functions.

3.1.3 Tools of the Trade

We receive information about the environment through our senses. The limitations of our senses dictate the parameters to our perception of the environment. We cannot decide what an environment is without also examining the capabilities of our own senses.

<table>
<thead>
<tr>
<th>Sensory Modality</th>
<th>Sensitivity/Resolution</th>
</tr>
</thead>
</table>
| Touch            | 10-100 micron vibration  
|                  | 1-2 mm spatial resolution |
| Smell            | 7 dimensions?          |
| Sight            | ~400-700 nm in the electromagnetic spectrum  
| | 10 minutes of arc at 6 metres |
| Sound            | 20 Hz to 20 KHz depending on the intensity  
| | ~10 dB to 120 dB |
| Taste            | 4 dimensions: salty, sour, sweet, bitter? |

Table 3.1 Common senses and their sensitivity/resolution.

The five commonly accepted sensory modalities taught at primary school level are touch, smell, sight, sound and taste (Table 3.1). However, there are more: interoception, proprioception and exproprioception (Lee, 1978). Proprioception is the ability to sense the position and movement of body parts relative to each other.
whilst exproprioception is the sense of body position in relation to the environment. Interoceptors indicate the internal state of the body, e.g. hunger, thirst, tiredness, whilst our vestibular system (in our inner ear) provides us with information to help us balance. It has been proposed that taste has four dimensions and arbitrary tastes may be synthesised with combinations of these primaries (Carlson, 1986). Similarly, it is possible that smell may have many dimensions (possibly as many as seven) and so it may also be synthesised. Predictably, the senses commonly stimulated by current VR systems have already been quantified more precisely. With the ability to pick out millimetre detail at 6 metres, it is unsurprising that most people are disappointed with the display technology used in current HMDs.

The resolution of our senses would be a good place to start when determining what information to use to represent our environment and at what accuracy, but it would also be short-sighted. By exclusively adopting the human perspective we will inevitably lose some of the environment's actual fidelity, although it would not be noticed until an unconventional view was attempted. For example, assuming the behaviour of another animal, such as a cat, will involve a different set of environmental properties in order for the participant to interact effectively. Regardless of the practicalities of this, it is important to realise that the senses of a human may be supplemented through various equipment such as infra-red night vision goggles. Robinett (1992) also notes that if sensors can detect phenomena that are imperceptible to human senses, they could be linked to display devices. This would mean that these imperceptible phenomena could be rendered visible, audible, touchable or otherwise perceptible to a human being. In a way, creating a synthetic sense.

If we restrict, for example, the modeling of the surface properties of an object to how things look in the visible spectrum, we will not be able to simulate it correctly when seen through night vision goggles. We may also wish to view the environment from another animal's perspective, e.g. a dog sees in monochrome, not colour, and its hearing is far more sensitive than our own, to name but two differences. Using the knowledge of our sensory abilities to aid the design of human-computer interfaces is
essential (Anderson, 1993; Caird and Hancock, 1993; Mon-Williams et al., 1993), but as a guide to modeling the environment it can be shown to be ultimately inadequate.

### 3.1.4 Virtual Environment Taxonomies

This section presents four different definitions/classification schemes for VEs. Each of them tackle the task at a different level and some are more detailed than others. The major points are presented here and comparisons drawn.

#### 3.1.4.1 A Conceptual Virtual Reality Model

Latta and Oberg (1994) have proposed a conceptual VR model which embraces Gibson's work. VR interface technology is viewed as integrating perceptual and muscle systems but it was noted whilst deriving this model that fully integrating these systems would be impossible due to the complexity of the human interface. So the model only examines some perceptual systems, not all. An operational VR system is seen as providing a computer interface to specific human perceptual and muscle systems for the purpose of allowing the participant to perform operations that would not be possible without aid, e.g. a flight simulator. The model's emphasis is placed upon providing an interface to perceptual systems, not on describing what the interface looks like.

The conceptual model consists of a human and a technical view of the VR system. The human view is interested in the physical and psychological issues of stimulating and detecting the actions of the participant, whilst the technical view is concerned with the environment.

"The environment provides the stimulus that creates sensation while the individual takes action through movement. The environment ... is the total space, both real and artificial."


The definition and integration of the real and artificial environments is viewed as defining the participatory experience. The mapping of the physical sensors and
effectors supports definition of the participant’s perception of the environment and their actions on it.

<table>
<thead>
<tr>
<th>Model Source</th>
<th>Artificial</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td>Model dynamically changes during the participation based on the actions of the participant or other events. Model database changes dynamically.</td>
<td>Direct</td>
</tr>
<tr>
<td>Constructed</td>
<td>Model is defined a priori as a fixed space and objects. Model database is static.</td>
<td>Sampled</td>
</tr>
<tr>
<td>Recorded</td>
<td>Time recording of the space of interface parameters.</td>
<td>Modified</td>
</tr>
<tr>
<td>Transparency</td>
<td>(relative contribution between artificial and space components to create the environment)</td>
<td>Recorded</td>
</tr>
</tbody>
</table>

**Figure 3.1 Confection of artificial and real environments.**

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>Direct</td>
</tr>
<tr>
<td>1:1 correlation between time in the environment and the participant environment.</td>
<td>(x,y) matching of the participant space and the environment.</td>
</tr>
<tr>
<td>Multiple</td>
<td>Distance</td>
</tr>
<tr>
<td>(nt) time modification between participant space and the environment.</td>
<td>(mz) distance scaling of the participant space and the environment.</td>
</tr>
<tr>
<td>Fixed</td>
<td>Scaled</td>
</tr>
<tr>
<td>(T) fixed time between participant space and the environment.</td>
<td>I(x, y, z) or (mx, ny, oz) scaling of distance for the spatial dimensions between participant and the environment.</td>
</tr>
<tr>
<td>Remapped</td>
<td>Functional</td>
</tr>
<tr>
<td>f(t) functional remapping of time between participant and the environment.</td>
<td>f(x, y, z) functional remapping of distance for the spatial dimensions between participant and the environment.</td>
</tr>
</tbody>
</table>

**Figure 3.2 Type of time and space.**
Technical confection and the real environment make the technical view of a VR system. Confecting is the process of preparing or making, especially by combining. Latta and Oberg believe that in VR we are confecting a participatory environment by combining a real environment with an artificial one (Figure 3.1). The technical confection includes a confection model that achieves interface control, defines the artificial environment and mediates between the participant and the real environment. A confection model can support independent models for each perceptual system. It also supports independent models for each muscle system, but the participant’s detection of the action is usually correlated with perceptual systems.

Figure 3.2 shows the ways in which space and time may be altered from their natural direct state to modify the experience. Latta and Oberg believe that there is a natural hierarchy in managing and controlling a VR system based on the parameters of the technical confection model. First the mapping of the sensors and effectors supports definition of the participant’s perception of the environment and their actions on it. At the next level the model source defines the static and dynamic aspects of the environment. Finally, space and time have equal importance: they are independent of each other but dependent on the first two levels of the confection model.

### 3.1.4.2 An Experience Taxonomy

Warren Robinett has proposed a tentative taxonomy to classify all varieties of technologically mediated experience (Robinett, 1992). This distinction is offered for “experiences”:

- **Natural experience.** Directly perceiving the properties or behaviour of something physically present before the perceiver.

- **Synthetic experience.** Perceiving a representation or simulacrum of something physically real rather than the thing itself.

There are nine dimensions to the taxonomy of a synthetic experience: *causality, model source, time, space, superposition, display type, sensor type, action measurement type* and *actuator type*. The first five dimensions deal with the
technological aspects of the devices used in the experience, whereas the last four are concerned with the sensor and motor channels used.

Causality refers to the way the VE is experienced, either via a previous recording or transmission, e.g. teleoperation, or totally simulated where actions in the VE have no effect on the real world. The second dimension states that the human user perceives a virtual world that is defined by a possibly changing database called the model. This model can be scanned, constructed, computed and edited. Both time and space may be either aligned, displaced, differ in scale, or be related by a distortion mapping. The time possibilities are 1-to-1 time-scale, accelerated (or retarded) time, frozen time and distorted time. Space may be registered, displaced or expanded (or miniaturised). The last technological dimension, superposition, basically refers to the possibility of merging the VE upon the real world, e.g. using a see-through HMD, or at another extreme, totally isolating the participant within the VE.

Display type and sensor type are the next two dimensions of the classification scheme. They present the potential for local input devices to be linked to remote devices in order to effect a change in the remote environment.

3.1.4.3 Multiple Environment Integration

A proposed definition of a VE, not visibly influenced by Gibson, is:

"A multi-dimensional experience which is totally or partly computer generated and can be accepted by the participant as cognitively valid."


Jense and Kuijper also view a VE as an integration of environments, in this case, three:

2. Physically modeled environment.
3. Real environment.
The computer-generated environment is created using a system consisting of sensor, control and actuator subsystems. The physically modeled environment contains objects that are also present in the real environment being simulated, e.g. a replica of an aircraft cockpit may be used to enhance a flight simulator. The real environment is
also viewed as an important component in a VE because it can stimulate senses that may be used to add realism to the simulation. However, the difference between the stimuli created by the physically-modeled environment and the real environment is not an easy distinction to make.

Another classification scheme is proposed based upon the amount of stimuli created by each of these three types of environment (Figure 3.4). A soft VE does not use any physical models to generate stimuli whereas a hard VE uses little else. Immersive VEs cut the participant off from the outside world and non-immersive systems use the real world in the VE. Most systems fall somewhere in between.

**3.1.4.4 Content, Geometry and Dynamics**

"... we can define virtual environments as interactive, virtual image displays enhanced by special processing and by nonvisual display modalities, such as auditory and haptic, to convince users that they are immersed in a synthetic space."


A formal definition for the environment, the theatre of human activity, is offered in Ellis (1991), which consists of three parts: **content**, **geometry** and **dynamics**.

The content of the environment is its objects, these are described by **state vectors** which are a description of the **properties** of the objects. **Actors** are similar to objects but may be distinguished by the fact that in addition to properties they have **capacities** to initiate interactions with other objects. The **self** is a distinct actor in the environment which provides a point of view from which the environment may be constructed. Anything outside of the self can be considered the field of action.

The description of the environmental field of action is called the geometry which has **dimensionality**, **metrics** and an **extent**. The dimensionality is the number of independent descriptive terms that are needed to specify the position vector for each element of the environment. Curved or straight lines are established through metrics which are systems of rules that are applied to the position vector. The extent is the range of possible values for the elements of the position vector. Following on from
this, the field of action can then be described as the product of all the elements of the position vector over their possible ranges. Kinematic constraints restrict the vast number of possible paths an object may take through the environment.

The dynamics of an environment are the *rules of interaction* among its contents. The transfer of energy or information that occurs during interaction alters the state vectors of the objects involved. All interactions can be reduced to binary interactions which may be ordered based on the ranking of the elements involved. Dynamical rules describe the result of interactions between the environments contents.

### 3.1.5 An Abstract Model

The definitions of a VE offered by Jense & Kuijper, Latta & Oberg and Robinett all acknowledge the integration of different types of environment, whilst Ellis places more emphasis on the human interface technologies. By evaluating the amount of real and physically modeled stimuli created by each of the three types of environment proposed by Jense and Kuijper, it is possible to classify VEs at a high-level and this provides a basis for comparison. Although a more detailed evaluation would be better undertaken using Latta and Oberg's classification model. Only Robinett and Ellis, however, recognise the importance of perspective on the environment. Even though our senses are limited, it may be desirable to simulate a wider bandwidth of information. This would permit the simulation of sensory-enhancing equipment or for the participant to view the environment in an unconventional way. In other words, each animal within the environment may have a unique view of that environment and hence is concerned with a subset of the environment's total properties. A preferable ecological definition for a VE would therefore be:

* A totally or partly computer-generated environment that contains enough information so that it may support affordances for different animals simultaneously.

Where an "animal" is an entity that could be a human with augmented senses, an object with some notion of artificial intelligence, or anything that has a unique perspective on the environment.
In order for a system to be able to support VEs of very different properties, it must have a flexible structure for modeling. Following the ARI decomposition of a model, the chosen abstract model of the VE may be represented in many different ways, each of which may have strengths and weaknesses. Each representation can also be implemented using many different methods, each having good and bad points. However, underlying all the possible implementations and representations should be a sound abstract model.

The model presented by Ellis is quite detailed and uses physics-based concepts to the point at which it could be confusing, at best, and restrictive, at worst, when considering a VE that does not behave according to natural physical laws. Ideally, the abstract model should provide a simple and flexible basis of representing any type of environment. The author believes that such a model exists in the basic structure of our universe and it is the model that should be used. A plausible description of this structure is:

A Universe contains all things that exist. These things may be described as Entities. An Entity consists of one or more Properties. An Entity may or may not interact with other Entities as dictated by Universal Laws. A Universal Law is an equation of constraint expressed using Properties, Universal Constants and other Universal Laws. A Universal Constant is a quantity that does not change throughout the whole Universe.

3.1.6 Representations

The use of terminology in the abstract model is meant to reflect its origins and not its possible applications, fortunately it fits quite well in the context of describing a VE. This model is, in fact, a very basic description of any form of structured data. A universe might be compared to a database: an entity is equivalent to a record, the properties are the record’s fields, the universal laws correspond to the relationships between records and so on.

Given that we have established a suitable abstract model for our VE, the next order of business is to find a suitable representation, something we can edit, manipulate and
generally play with until we are happy that we have a description that embodies our ideas. In essence, a specialised data description language. This task is undertaken in chapter 4 but before language design is examined, we should first consider modeling methods.

An entity modeled using the abstract model detailed above may have many different representations. There may be a visual representation, an aural representation, tactile, thermal, etc. Each of these is interested in a number of properties, some are shared between them and often some are unique to the representation. They are all governed by a subset of the total universal laws and are applicable to a subset, if not all, of the entities in the Universe.

One possible solution would be to model these representations independently, but this can introduce a great deal of data redundancy. For example, the physical appearance of an entity would only seem to be of interest if you are building a visual model. However, a tactile model is also heavily based on the geometry of the entity and, of course, how the entity distorts sound is based on geometry as well as other factors (Astheimer, 1993). If the shape of the entity changes then the relevant properties in the other models would also have to be changed. A shared structure of information would therefore seem appropriate, at least until design decisions for the implementation of these models need to be taken and then we are faced with the time old battle of distribution versus replication.

It is at this point that we should also consider the design process. Without doubt modeling, whether it is geometrical or mathematical, can be as time consuming as developing the code to execute it, if not more so.

3.2 The Modeling Process

How the information in the model is organised and shaped into the final form is not just dictated by the thing being modeled, but how the model is derived. This section takes a cursory glance at the possible approaches to actually building a model and
their effect on the design process. Consideration of these factors aids the design of
the modeling language (representation) and the supporting system.

There would appear to be three levels of "reality" (for lack of a better term) that can
be created:

- An observer-oriented reality would provide adequate simulation of the inputs
  and outputs required by a human at the required accuracy.

- An environment-oriented reality would provide adequate simulation of all
  inputs and outputs affecting the environment\textsuperscript{1} at the required accuracy.

- A universe-oriented reality would provide adequate simulation of all inputs
  and outputs at the highest possible accuracy.

All current systems cater (in one way or another) for the first category, an observer-
oriented reality. Most image generators only model the attributes of a surface which
are acted upon by visible light. Few give consideration to the rest of the
electromagnetic spectrum, e.g. ultraviolet, infra-red, radio etc., because it is not
generally required. In the same way, acoustic systems only deal with the range of
frequencies that we can hear, even though many others affect us, e.g. ultrasound. The
technology for the simulation of stimuli for smell, taste and touch are only just starting
to be developed but clearly an entity's complete set (or subset) of properties must be
modeled to permit their use. These observer-oriented systems also fail to easily
accommodate simulation of things that do not directly affect us but we wish to
visualise, e.g. the path of radio waves, infra-red light, and so on.

A universe-oriented reality is the ultimate goal and would model everything in fine
detail and without exception. In this context, "universe" is intended to mean the thing
that is being modeled, in its entirety. Of course, it is possible that the amount of
processing power and storage required to simulate the universe would exceed its size

\textsuperscript{1}In this context \textit{environment} means the volume of entities surrounding the participant. The size of
the volume is arbitrary.
in the first instance! Nevertheless, it should be considered as one of the ultimate goals of a VE system, however vain.

The next best thing would be an environment-oriented reality where the microcosm would possess a subset of the properties of the universe. These would be simulated to a high enough level of accuracy to allow their examination and a more accurate and realistic simulation of the participant's environment. How big the environment should be is a good question. Probably any volume that does not encompass the universe could be modeled in this way.

3.2.1 Model Construction

Some attempts have been made to provide higher-level modeling systems (Hemmje & Strohmer, 1993; Luciana et al., 1991; to name a couple), but these still concentrate on a particular type of information or specific application and are not applicable to the general task of modeling a VE. There would seem to be two basic approaches:

1. Take a very general, flexible and computationally expensive model and simplify/remove the parts that are not relevant to the case in hand.

2. Take a skeleton model and then build on it, successively specialising and tweaking.

Both of these methodologies can be seen to use a hierarchical approach in different ways. Using the first method, the designer is given the most complicated model that can be described and then they selectively remove/simplify the parts that are not relevant for the intended simulation. Each branch of the tree would therefore represent a progressively simple subset of the general model. Method 2 does just the reverse and could be likened to the object-oriented language feature of inheritance. Take a simple abstract class that provides the basic structure and fabric of a VE and then derive classes from it that provide it with some "flesh". Each new derived class would increase the realism of the simulation and also its computational complexity. Each of these approaches is valid and may be compared to the programming design methodologies of bottom-up and top-down design respectively.
3.2.1.1 Methodology Choice

However, method 1 requires a lot more initial work because a great deal of consideration needs to be given to all of the simulation's goals and requirements. This is a potentially impossible task and, if anything, it will be limiting. Its advantage is that little or no work needs to be done to the model to get a fully working simulation running. Unfortunately, the same cannot be said for the entity descriptions themselves, each one must have all of its parameters meticulously evaluated and initial values found.

Method 2 requires that a small extensible structure is derived and represented and thus provides the most flexibility. Its disadvantage is that the model's representation has yet to be created which, depending on the simulation, may take some time. The main advantage is that with smaller models there will be less preparation needed for the entities - only those parameters needed by the model will be evaluated. The universe could be represented as being composed of sub-universes or microcosms, each of which has its own laws to govern. Entities in each of the microcosms will possess enough properties such that the microcosm's laws may determine their behaviour. But what would happen if an entity from one microcosm would wish to move into another?

![Diagram](image)

Figure 3.5 Universe hierarchy tree showing possible entity migration paths.
3.2.1.2 Entity Migration

The issue of entity migration really only exists in the second modeling paradigm. With method 1, migration would just mean using a different subset of the entity's properties and would require little or no intervention on the part of the designer. However, using method 2 the microcosm that the entities immigrate to must have sufficient laws to govern them correctly. If it is to do the same job as the emigrated microcosm then it must possess its laws and properties, this would mean that the emigrated microcosm is a specialisation of the higher level. This sort of migration may be thought of as vertical, up the inheritance tree (Figure 3.5).

It is equally likely that an entity will want to use horizontal migration, i.e. moving from one microcosm to another, each with a common ancestor. The implication of this action is that some of the entity's properties will be shared, some will be left behind and others will be gained when entering the new microcosm. One logical course of action is to assign default values to these new parameters, although in practice this is unlikely to be a very satisfactory solution. Unfortunately, without some insight into the "purpose" of the entity, little else can be done automatically.

3.2.1.3 Modeling Process Summary

It would therefore seem sensible to use method 2 to develop VEs because it requires less initial work and presents a clear structure to the designer. At a reduced level it would be possible to simulate method 1 by redefining inherited laws, etc., to be simpler. Program design is often a combination of both bottom-up and top-down so it would seem reasonable to expect a similar approach to VE design.

3.2.2 The Design Process

All of the systems reviewed in the previous chapter treated the modeling process as an independent task that is performed initially, the result of which is executed. Some systems permit minor modifications to the model to be made at run-time, but this is usually limited to changing the values of selected properties, e.g. entity colour. An
entity may leave one VE and enter another through migration but major changes to
the entity itself or its environment are not possible.

Such changes may be the addition of new entity properties, the alteration of a law
governing those properties, changing the value of a constant, etc. The ability to
change the VE at run-time has several advantages:

1. Development. By integrating the modeling and execution phases a prototype
model can be refined and extended into the finished product without stopping
the simulation. Whereas existing systems require some description to be
written in a language, compiled/interpreted and then executed, integrated
development tools would remove this distinction.

2. Experimentation. A better development environment will encourage
experimentation in the form that the VE takes. This is, of course, currently
possible but the time cycle is large enough to become frustrating. A friendly
modeling system increases the likelihood of better VEs and, hopefully, better
designers.

3. Evolution. The ability to modify the VE need not remain in the hands of the
designer. On a restricted level it could be given to the participants in the VE
or, more interestingly, to the entities themselves. This reflects an animal's
ability to influence its environment, especially true in the case of humans.

### 3.3 Real-time Virtual Environment Displays

So far, this chapter has examined the somewhat abstract topic of VE modeling.
Consideration has been given to our natural environment in the hope that it will add
some insight into what we are trying to achieve when modeling a VE. A good VE
should be intuitive to use; in other words the participants should have no trouble
navigating around the environment, interacting with it, and completing any task that
they set out to achieve. However, a sound VE model in itself will not achieve these
goals. Unless interaction is effortless (or "natural") then even the most detailed
model, built using the most advanced techniques, will fail to deliver the experience intended by the designer. This quality assessment is made via our senses and perceptual systems. If our perceptual systems are working normally then our energies will be expended on the task at hand. However, if we are fed information that disrupts the natural processes of our perceptual systems then we will either become aware of this problem or our performance will suffer. Therefore it is just as important how the environment is displayed, as the type of information contained within it. To understand the potential problems with current VE displays we must first establish the cause.

The purpose of a display is to take raw information from the environment, process it, interpret its meaning, and then present it in a form that enables the viewer to extract some meaning. A suitable practical example is that of a visual display which is driven by a CIG - although aural or tactile would also make good examples. A CIG must process the geometrical information in the model, including lighting, surface texturing, etc. The more information it processes, the longer it takes to complete the rendering. If this display is presenting the participant's view on the VE then the time taken to render the view may well depend upon where the participant is looking. Since the viewer will make decisions based upon the information the displays show them, e.g. what they see, then it is important that things appear where they should, when they should. Unfortunately the time between requesting a rendering of a new view and actually seeing it can be relatively large. The same statement can be applied to all types of displays, each of which may perform at different rates. This is not a situation which we have to tolerate when interacting with our natural environment. Consequently, at the very least, the viewer is presented with incorrect information for any given moment in time and, at worst, interaction with the VE is impossible.

The remainder of this chapter examines in more detail why the update rate of VE displays should be constant. In order to present the two possible solutions to this problem it is necessary to consider the workings of a VE system in a little more detail than before. This discussion is a precursor to the detailed system design described in the next chapter and clarifies one of the primary system requirements.
3.3.1 Problems with Variable-Rate Systems

In this section we present an example of the effects that a variable update rate has on interactivity. This is quantified by the application of a visual perception theory. The other benefits of a constant update rate are also discussed.

3.3.1.1 Display Artifacts

Consider a virtual ball moving straight towards you at a constant velocity of 1 m/s. It starts its journey 10 metres from you and you are attempting to catch it. Let us assume that a simulation of this will use a typical variable-rate CIG and a monitor (showing the catchers view) with a refresh rate of 60 Hz. When the ball is in the distance and hence quite small, the CIG manages to generate a new frame 30 times a second. This means that every 2 monitor refreshes a new picture will appear.

If the CIG maintains this frame rate then the velocity of the ball will indeed be constant. However, if the CIG should manage to complete its work within a 60th of a second then the ball’s velocity will appear to have doubled to 2 m/s! On the other (more likely) hand, if the CIG’s workload takes longer than 33.3 ms to complete and hence only produces a new frame every 3 monitor refreshes, then the velocity of the ball will appear to reduce by 1/3 to 0.66 m/s.

If the frame rate was to go up or down each time an image was being rendered then catching the ball will be made more difficult. In this case we are likely to see a drop in update rate because as the ball comes towards us, it expands. If the ball was textured and the background blank, this would mean that there are more pixels to fill and hence more work to do. Certainly, we are not seeing what the designer of this simulation wanted us to see.

---

2 The word "render" is used in this thesis to embrace both of the classical geometrical and rendering stages used to produce an image.
Another more practical example is that of a driving simulator. Given the task of following a vehicle and ensuring that you do not crash into it would be made difficult if the vehicle would seemingly slow down and speed up quite uncharacteristically.

3.3.1.2 Judging Time-to-contact

Lee (1976) presented the Tau theory which suggests that our ability to judge our time to contact with a given target is based upon the rate of expansion of the target on the retina. This may be applied to our ability to catch balls as well as how we control our deceleration, among other tasks (Lee, 1993).

The time-to-contact (TTC) of the virtual ball may be expressed as:

\[
TTC = \frac{\text{Distance}}{\text{Velocity}}
\]

Figure 3.6a shows the TTC assuming that we maintain a constant update rate of 30 Hz which gives us a perceived constant velocity of 1 m/s. The impact of a variable update rate is shown in Figure 3.6b. Each time the update rate changes so does the TTC, forcing the catcher to continuously readjust. In this case, the catcher will probably catch the ball because the update rate has slowed down so much that the perceived velocity of the ball at 5 Hz is 0.16 m/s, making the task trivial. They are unlikely, however, to be using TTC information to help them catch.

3.3.1.3 Affects on Latency

If the time between sampling input devices and updating the display is too long it can contribute to simulator sickness (Pausch et al., 1992). Just how long is too long is not clear, additionally it is not clear whether the systems used provided a constant or variable display update rate. There is evidence to suggest that humans can adapt to a constant degree of lag (providing that it is not too great) after a reasonable period of time, but how effective the interaction is depends on the task being performed. If the lag varies then adaptation is less likely and it is possible that this will add to simulator sickness.
Figure 3.6 The effect of update rate on time-to-contact (TTC).

a) with a constant update rate TTC decreases correctly at a fixed rate (top); b) a variable update rate causes a continuous readjustment of TTC (bottom).
3.3.1.4 Predictive Techniques

There are methods for reducing the impact of lag on the participant. Kalman filters can be used to compensate for the effects of lags within the system (Friedman et al., 1992; Liang et al., 1991; Dunnett et al., 1995). Such filters have been used to predict the movement of 6 d.o.f. sensors attached to parts of the body, e.g. head and hands. In the case of Head-Mounted Displays this means that the CIG can be asked to generate an image of the participant’s viewpoint a short while in the future such that the image reaches the display at the right time. The effectiveness of these filters relies on the constancy of the lag and hence the update rate, without it the results of the filtering would be meaningless.

If the progression of time happens at a known rate it is also possible to ensure that entities within the VE appear at their correct positions when the image is eventually displayed. This is especially useful when trying to compensate for the single frame delay introduced in double-buffered CIG systems.

3.3.1.5 External Device Synchronisation

It may also be desirable to synchronise the VE display with an external data capture system. An example of such a device is the Ober/2\textsuperscript{TM} infra-red eye-tracking system (Permobil Meditech AB, Sweden). A lot of effort has been expended by the manufacturers to ensure that a fast, constant sample rate is achieved, to such an extent that the host machine is configured solely for the purpose of controlling the eye-tracker. Sample rates over 1000 Hz may be achieved although 180 Hz is sufficient for tasks monitoring basic eye movements (Permobil, 1993). In order to determine where the participant was looking within the VE display requires the meshing of two data sets, each with a different sample rate. Whilst, on a variable-rate system, it would be possible to record the update rate and then fit the eye-tracker data set to this, the result would be an uneven spread of data points over time. With a constant update rate system, the eye-tracker rate can be set at a multiple of the update rate which makes meshing much easier and produces a consistent number of data points per second.
3.3.2 The Variable-Rate Paradigm

A typical simulation processing cycle is:

1. Sample input devices.
2. Perform dynamics calculations.
3. Update output devices.

The VE system may consist of many components, both software and hardware. With each component comes a response time, a best and worst case for receiving data, processing it and outputting a result. Exactly where the bottleneck in the system is depends on the nature of the VE or application. Typically the bottleneck is the CIG. This is especially true in low-end systems where the CIG is more (or totally) dependent on the host processor to complete its task. In this case, image generation often has to be scheduled along with input/output device handling and the dynamics calculations. It is also quite typical for the workload of each component to vary. This is especially the case in the CIG where scene complexity may vary drastically (Airey et al., 1990).

3.3.3 The Fixed-Rate Paradigm

In order to provide a constant update rate there are two possible approaches:

1. Derive some predictive algorithms that will enable us to determine the workload of each component and thus the system as a whole.
2. Restrict the update rate to the worst-case.

Both these methods are working to complete the 3 steps in our simulation cycle before a given deadline. Once this deadline has been met it is recycled and used again for the next VE display update.

If we adopt the first approach then we may use the knowledge of each component's performance to degrade the services it offers such that the deadline for each component will be met. Alternatively, we can demand less of the system such that, even in the worst-case, it always meets its deadline. This inevitably means using some
components at less than optimum performance. Both of these techniques will now be discussed in further detail.

3.3.3.1 Service Degradation

This technique requires a scheduler to determine acceptable time-frames within which each component in the system must complete its calculations. The addition of a scheduler brings us one step closer to a real-time system. Failure to meet a deadline will have different consequences depending on the application. A visualisation may be content with simply providing a lower update rate (albeit constant) whereas a highly interactive application may treat failure to meet the deadline as a fatal condition.

It should be noted that some systems have decoupled the rate at which component services are requested and the update rate of the CIG (Shaw et al., 1992; Wloka, 1993; UVa, 1995). Therefore the simulation may progress as fast as possible, while the CIG generates images as fast as it can.

However, CIG performance can still benefit from service degradation. Holloway (1992) draws as much of the visual scene as possible whilst still attempting to meet the deadline. To achieve this, the Viper system uses a special feature in the Pixel-Planes Programmers Hierarchical Interactive Graphics Standard (PHIGS) implementation which allows traversal of a particular part of the database hierarchy to be terminated based on a conditional check of a global flag. In addition, visual objects\(^3\) were given either a high or low priority. High priority objects were always drawn and low priority objects only if time allowed. There is no guarantee that the image will be rendered within the allotted time since Viper uses successive estimates to decide whether it has enough time to render any more and is at the mercy of the underlying operating system (OS).

\(^3\) The visual component of an entity.
Wloka (1993) proposes a system for time-critical graphics which uses knowledge of
the dynamics behaviour of the simulation and a modified graphics database model
combined with a scheduler to implement this technique.

As Wloka notes, few CIGs support service degradation techniques. The nearest
facility that most provide is Level Of Detail (LOD) which attempts to reduce
workload by automatically substituting models of different visual complexity based on
distance or screen pixel coverage (Reddy, 1995). SGI's IRIS Performer™ goes one
step further by providing a mechanism known as dynamic LOD scaling. This provides
enough basic information for Performer to decide which combination of LOD models
will complete rendering within a certain amount of time (SGI, 1995). The other work
done in this area is at the application level as opposed to adding functionality to the
CIG. Airey et al. use LOD along with other pre-processing techniques to support an
adaptive refinement system that trades image realism for speed. Funkhouser and
Séquin (1993) use cost and benefit heuristics to determine which LOD model should
be used. The cost of an object is the time it takes to render an object with a given
LOD using a certain rendering algorithm, whilst the benefit is an estimate of the
contribution of the model to human perception. Encouraging results are obtained
using this approach, however, even this technique is not sufficient to cope with
extreme cases such as changing the view from looking at the sky to looking at a fully
textured model of a town.

3.3.3.2 Worst-case Operation

Establishing what the worst-case is for a given VE can be accomplished by either
working out by hand the worst performance of each component or by “exercising” the
VE over a period of time. The latter method is very convenient and relatively
effortless to perform, however its effectiveness is dependent on exercising the parts of
the system that will present the worst performance, either on their own or combined
with other components.

A major advantage of this approach is that it may be used on systems without real-
time extensions and although scheduling still plays an important part, it is done on a
decidedly pessimistic basis. The price paid for this type of predictability is the under utilisation of the available services, which is sometimes quite extreme if there is a large bottleneck in the system.

3.3.3.3 Implementation

Regardless of which method is chosen, the control of the CIG is the same and the possible scheduling options limited. A prototype implementation of the ideas presented here is given in chapter 5.

3.3.4 Conclusions

Producing successive displays of a VE at a variable rate can be shown to cause interactivity to suffer. The sense of presence in VEs is another area where variable rates may have an effect. In the study performed by Barfield and Hendrix (1995), five different update rates were used to examine the sense of presence. Efforts were made to ensure a constant update rate but it would also be interesting to see the effect that a variable update rate has on presence - which is currently a far more realistic situation.

Increasingly, other complex standalone hardware (such as eye-trackers) are being incorporated into VE systems. Without a common time-frame, attempts to synchronise this equipment with a VE system can produce anything from erroneous to useless results.

Whilst a constant update rate permits object positions to be calculated into the future, predicting the actions of a human interacting in the VE is another matter. Estimation of the participant’s head and possibly hand movements may be accomplished using Kalman filtering, but there is no way of anticipating what they will do next. Because of this there will always be a latency between human action and displayed reaction with an order of one or two updates. However, it is surely better to base a judgement on a VE whose state is correct for that moment in time, than to base judgements on out-of-date information.
3.4 Summary

Attempting to define a VE in one sentence is next to impossible but its most important features can be expressed concisely. The general consensus is that a VE is actually far from virtual. It is a combination of our natural, physical environment and the computer-generated environment that is presented to the user through a wide variety of displays. It would seem that Virtual Environment is not a suitable term for describing such a phenomenon, Artificial Environment or Synthetic Environment would probably be more appropriate. However, almost all of the literature talks in terms of VEs so it would seem sensible to stick with the most commonly used term.

After establishing exactly what a model is in the general sense of the word, the search for an abstract model began. It is clear that an environment is perceived differently depending on the viewer’s perspective. This change in perspective may be due to the augmentation of our natural senses or even a change in species. Additionally, in order to enable unconventional input devices and displays, e.g. tactile, to function correctly, it is necessary to model more information than usual. Consequently, a method of modeling the diverse information present in the environment must be found.

As a prologue to finding a suitable representation for the proposed abstract model (presented in chapter 4), the modeling and design processes were considered. A hierarchical approach using inheritance to extend and specialise successive models was the favoured modeling methodology. This increases the flexibility available to the designer and, with the correct system support, will hopefully aid them in the production of better VEs.

After dealing with rather esoteric issues, a more down to earth problem was discussed. Variable update rates can destroy the visual illusion because this effect is not experienced by us when interacting with our natural environment. The synchronisation of audio with visuals can also fall victim to such a situation (as will most displays). Correct interpretation of data from input devices can also suffer in systems that have a variable duration between device sampling and display output. The technical details of a constant-rate visual display are presented in chapter 5.
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.

<table>
<thead>
<tr>
<th>System</th>
<th>Theoretical Bandwidth</th>
<th>Class</th>
<th>Medium¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>V.34 Modem</td>
<td>28.8 Kbps</td>
<td>Dedicated link</td>
<td>Copper telephone line</td>
</tr>
<tr>
<td>ISDN</td>
<td>64 Kbps per channel</td>
<td>Dedicated link</td>
<td>Copper telephone line</td>
</tr>
<tr>
<td>Frame Relay</td>
<td>56 Kbps - 1.98 Mbps</td>
<td>WAN</td>
<td>Coaxial</td>
</tr>
<tr>
<td>Ethernet</td>
<td>10 Mbps</td>
<td>LAN</td>
<td>Coaxial or twisted-pair</td>
</tr>
<tr>
<td>Fast-Ethernet</td>
<td>100 Mbps</td>
<td>LAN</td>
<td>Coaxial or twisted-pair</td>
</tr>
<tr>
<td>CDDI</td>
<td>100 Mbps</td>
<td>LAN - MAN</td>
<td>Fibre-optic</td>
</tr>
<tr>
<td>FDDI</td>
<td>100 Mbps</td>
<td>LAN - MAN</td>
<td>Fibre-optic</td>
</tr>
<tr>
<td>ATM</td>
<td>155 Mbps+</td>
<td>LAN - WAN</td>
<td>Fibre-optic</td>
</tr>
</tbody>
</table>

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

¹This 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 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.

Figure 4.1 A taxonomy of real-time scheduling algorithms.
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. 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

---

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.
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 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 optimisation 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 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.

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

§ 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 4.4.3.

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 structure at run-time. None of the languages reviewed enable the data structure to be altered online 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.

\(^2\) That is not to say that it would be inefficient (Templ, 1993).
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: CONSTANTS, PROPERTYs, CONVERTERs, FUNCTIONs and other ELEMENTs. 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
 |
figure : 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 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 \texttt{Update} function. It is also used as a focus for the synchronisation of the data within the simulation.

\begin{verbatim}
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;
}
\end{verbatim}

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: \texttt{REAL}, \texttt{INTEGER}, \texttt{STRING}, and \texttt{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).

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

### 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.
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.

```
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.

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.
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.
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.

```plaintext
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.

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 above 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 Figure 4.11 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.

```uml
// 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.**

### 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.

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 a 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.

![Figure 4.14 Example structure of a Universal Simulation System.](image)

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.

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.

![Diagram showing the organization of a USS with populated USNs]

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

### 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.

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.

![Diagram of USS hierarchy](image)

**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 \textit{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 \texttt{Update} function, sends any modified state back to its local UM and waits for the next update message.

![Diagram of simulation update process]

\textbf{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 \textit{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).

![Possible communication path taken by a message sent from an entity to all managers.](image)

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.

![Diagram](image)

**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:

\[
\text{ID size} = \text{number of entities system-wide} \times \text{number of entity's dependencies}
\]

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 a 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.

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 \textit{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 special managers 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

151
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 need is during an entity migration, at which point a migration list is maintained. This list 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 UM's 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 where 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 for efficiency and the RM would be informed that this resource is no longer available since it 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 nodes 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. \texttt{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 is 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.
This chapter presents the implementation of a prototype USS based on the design described in the previous chapter. A full implementation of the design would take a considerable amount of time, far in excess of that available to the author. Therefore only those components (or parts thereof) that were required to demonstrate the architecture’s key points were implemented.

The feasibility of implementing a scheduler on top of a general-purpose operating system is explored with the implementation of a solution to the real-time VE displays problem. This is followed by a description of the platforms upon which the prototype was designed to run.

The USS implementation details begin with an examination of networking in a heterogeneous network, proceeded by configuration control and an implementation of a UML interpreter. Following details on each system component, the chapter concludes with a list of improvements that can be made to the prototype.

### 5.1 Real-Time in the Real World

The QNX operating system (QNX Software Systems Ltd., Ontario) was used by the author to develop the VE Support System (VESS) for experimental work undertaken
in the VEL, University of Edinburgh. QNX is a Portable Operating System Interface (POSIX\textsuperscript{1}) compliant, multi-tasking, distributed, real-time operating system (OS). It provides a priority-driven, preemptive scheduler which is certainly suitable for a soft real-time system and with great care can be used in a system with static hard real-time constraints. Part of VESS's functionality was enforcing the constant update rate of the CIG displays. The implementation of this solution is presented in this section and was used to explore the viability of implementing a scheduler-based prototype USS.

Section 4.3 presented a taxonomy of real-time scheduling algorithms. In the field of VR, many systems claim to be real-time and can indeed be classified as soft real-time systems. The service degradation option for ensuring a constant VE display update rate discussed in section 3.3.3.1 requires a deadline scheduler. Unfortunately, implementing such a scheduler on top of a normal multi-tasking OS such as UNIX is problematic. Most OSs are not suited to real-time purposes, i.e. they do not provide ways of guaranteeing response times for certain events such as interrupts, IPC and disk I/O, etc. Those real-time systems that do provide such guarantees often use static schedulers. A VE system is dynamic and therefore a scheduler is required that can also cope with changing existing deadlines and the introduction/removal of new tasks. Since a dynamic deadline scheduler was not available it was decided to adopt the worst-case operation solution (section 3.3.3.2).

### 5.1.1 Real-Time Displays

There are a number of operations and pieces of information that a visuals manager needs to enforce a fixed frame rate in the CIG:

1. Manual control over buffer swapping
2. The time between one display refresh cycle and the next.
3. The amount of time that the rest of the system components need to complete their work for the next simulation update.

\textsuperscript{1} The 'X' would appear to have been added to reflect the fact that the interface is based heavily upon the UNIX variants.
5.1.1.1 Manual buffer swapping

This is essential to the task at hand. Double-buffered systems will display the last rendered image until the current one has been finished. At this point the new image is displayed and the next image is rendered into the other buffer. The switch actually happens during the next vertical retrace (or flyback) phase. On displays such as monitors, this is when the electron gun makes its way from the bottom-right corner of the tube (as the viewer sees it) to the top-left, ready to start drawing the next picture. To achieve a constant frame rate we must be able to choose which vertical retrace is used to switch display buffers.

5.1.1.2 Inter Refresh Time (IRT)

The IRT is the time it takes to draw one picture on the display including the vertical retrace period. For example, say that a 640x480 resolution image is refreshed at 60 Hz. This means that the IRT is 1000 / 60 = 16.66 ms. The refresh rate varies depending on the resolution of the video signal, e.g. an 800x600 pixel image is often refreshed at 72 Hz, and different display devices can handle different ranges of refresh rates.

The refresh rate may be provided as a parameter at run-time or, alternatively, this information may be obtained from the CIG which is the approach adopted here. Each time the CIG generates a vertical retrace it also generates an interrupt which is intercepted by the host machine and the time stored. The next time an interrupt is caught, the time difference is calculated and this gives us the IRT.

This technique will only work if the host machine has a clock that can provide nanosecond accuracy and the interrupt latency\(^2\) is bounded. The latter point is by no means certain in non-real-time operating systems such as UNIX and was one of the main reasons QNX was used.

\(^2\)The time between the interrupt being generated and the process on the host machine being notified of the event.
5.1.1.3 Inter Update Time (IUT)

The total processing time required for one simulation update is provided by the scheduler and the IUT is the nearest multiple of the IRT to the given time. In other words, the total work time can be expressed as a number of display refreshes. For example, if the IRT is 16.66 ms and the work takes 40 ms, the IUT would be 49.99 ms, i.e. the work may be done within 3 refreshes of the display.

5.1.1.4 A comparison of paradigms

Figure 5.1 shows the various ways of scheduling the work to be done each frame. There are three basic stages: calculate, render and display. Figure 5.1a shows how these stages fit together in a variable-rate system and how they relate to the display refresh cycle. The time at which the frame may be displayed varies and rarely coincides with a vertical retrace, which means that the actual buffer swap happens sometime during the next cycle. As shown in the diagram, most of the time the calculation stage may progress immediately and by the time this is finished, the buffers have been swapped and the render stage is ready to continue. However, the last complete cycle in Figure 5.1a shows that it may be necessary for the render stage to wait until the buffers have been swapped. This is because the buffer that will be filled next is currently being displayed.

The scheduling of the work in a fixed frame rate system is shown in Figure 5.1b. The time between the end of the rendering stage and the display will vary depending on how long it takes to render the scene. Pseudo-code for this process is given in Figure 5.2.

Both these examples assume that all work is being done by one CPU. If the image generation can be dedicated to another CPU or the system is equipped with a separate graphics subsystem, then time may be saved by scheduling the calculate and render stages such that they overlap as shown in Figure 5.1c. This is best achieved by starting the redraw as soon as possible (since it will take the longest time to complete). In order that we are rendering the most up-to-date state possible, the calculation stage is done before the end of the previous frame. By performing these
two stages in parallel it also means that more time can be spent on the simulation dynamics. Obviously, failure to complete either of these stages before the designated refresh occurs is a system failure.

Figure 5.1 Simulation cycle scheduling.

Regardless of technique, it is important to understand how the CIG works and the latency that it introduces into the process since not all CIGs work the same way. For example, an SGI RealityEngine/2™ introduces a one frame latency whilst the Real World Simulation Reality3™ PC card produces a two frame latency. The latter system was used in this implementation and, to compensate for this latency, state calculations must be done two updates before the image needs to be displayed.
// Step 1: Initialise key variables
Calculate IRT
Calculate IUT based on totalWorkTime
Enable manual buffer swapping
displayTime = 0

// Step 2: Synchronise loop with display
Wait for refresh

// Step 3: Enter main processing cycle
While simulation not complete {
   // Step 3.1: Calculate state
displayTime = displayTime + IUT
Calculate state of VE for displayTime

   // Step 3.2: Draw new image but don’t display
Redraw display

   // Step 3.2: Display image exactly on time
Wait for end of IUT period
Swap buffers
}

Figure 5.2 Pseudo-code for the fixed frame rate, worst-case simulation cycle.

This method of controlling double-buffering can be applied to most CIGs with few problems since it utilises existing functionality. It may be necessary, however, for the API to be modified to gain access to this functionality.

5.1.1.5 Further improvements

It is quite common for the render stage (even in its worst-case) to complete before the time that the display stage needs to run (as shown in Figure 5.1c). If this is the case then the start of the state calculation, which includes input device sampling and the render stage, may be put back such that there is even less delay between calculation and display (Figure 5.3a).
A more advanced technique is the controlled increase or decrease of update rate. It would be possible to detect whether the CIG is capable of going faster, e.g. making the change between 30 Hz and 60 Hz, by maintaining a history of its execution time for each update. If, after a small period of time, this new potential performance was sustained then the other stages could be rescheduled, if possible, and the switch made (Figure 5.3b). In a similar way, by monitoring the performance profile, a slow increase in workload could be detected and a decision made to extend the deadline. Once a decision is taken to change the deadline, no further changes must be made for a reasonable period of time, e.g. a couple of seconds, or things would quickly degenerate into a variable-rate system. Such an enhancement could also help overcome the fact that the worst-case approach assumes that the environment is quite static and does not cope well with the dynamic creation or destruction of objects.

Some multiprocessor CIGs already monitor image complexity to aid in processor load balancing. For example, the Reality3™ system, uses knowledge of the changing complexity of each scan-line to predict the load distribution for the next update. With
additional functionality in the API, these calculations could be used in the decision-making process. It is true that simple decision-making logic could be flawed by fast increases or decreases in workload, but the potential increase in system fidelity makes it worthy of more investigation.

A deadline-based approach also provides the framework for the application of object priority systems within the CIG as well as the visuals manager. Objects may be drawn, partially drawn or skipped depending on their priority (as in Holloway’s Viper system).

5.1.2 Conclusions

The problem of presenting a temporally correct view of a VE has implications throughout the whole support system architecture. The most important (and often the most expensive) component of a VE system is the CIG. Most CIGs provide some kind of service degradation in the form of LOD (section 3.3.3.1), but this is insufficient and improvements must be implemented via the API.

The implementation presented above has been used effectively over a number of years in the VEL. However, its utilisation is not as simple as “plug and play” since its performance is highly dependent on the other processes used to simulate the VE. For example, if data logging is added to the simulation then this introduces an execution path that passes through the filing system manager and the hard disk device driver. Each of these processes have their own timing constraints, are dependent on a number of interrupts and must therefore be accounted for in the schedule. Other changes that can have large effects on reliability are: communicating with a machine via the (dedicated) network, increasing the complexity of the visual database being used, adding another input device, synchronising with an external device, etc.

Even under QNX, which supports POSIX 1003.1b Real-Time Draft Standard Process Scheduling, getting an application to schedule every component to meet worst-case deadlines can be quite time consuming. The possibility of doing the same under a heavyweight OS such as System V Release 4 UNIX is very low. In addition, general
OSs use virtual memory and have unbounded interrupt latency to name but two confounding features. Since it was the intention to demonstrate USS running on different machines and operating systems (albeit UNIX variants), it was decided not to attempt a real-time implementation.

5.2 Target Platforms

From the outset it was intended that the prototype should be portable to a number of different platforms. It was planned to use QNX during initial development; so it was a natural progression to use other platforms with similar operating system functionality, preferably with some POSIX compliance. These platforms are briefly described in this section whilst specific details are dealt with in section 6.2. The choice of an Implementation Language, IL, is also discussed.

5.2.1 IBM Personal Computer Compatibles

Three PC compatibles on a dedicated network within the VEL were available to the author, each running QNX. One of these machines acted as a gateway to the Internet thus opening up the possibility of connecting multiple USSs on a heterogeneous network. Each machine had between 16 and 24 Mbytes of main memory and ranged in power from an Intel 486/50 MHz to an Intel Pentium/90 MHz. The memory capacity is important because QNX does not use virtual memory. Additional resources included a dedicated CIG and sound generation equipment.

5.2.2 Cray T3D

Originally it was intended to use the Edinburgh Parallel Computing Centre's (EPCC) Cray T3D super-computer as the second platform to run the prototype. The T3D was installed with 160 nodes, each with 2 DEC Alpha 21064 processors running at 150 MHz and 128 Mbytes of memory (64 per processor). The T3D is connected to the real world via a Cray Y-MP host running UNICOS, a POSIX compliant OS. Unfortunately use of the Cray had to be abandoned for a number of reasons:
1. Despite having an 8 MByte "microkernel", no IPC mechanism is provided - only shared memory operations are available. To ease this problem, three messaging libraries are available:

a) Portable Virtual Machine (PVM - Geist & Sunderam, 1991). This library makes use of a central server process which runs on the T3D host. Unfortunately the central server process does not fit with the USS design.

b) Message Passing Interface (MPI, 1993). This is an attempt to standardise on an IPC mechanism incorporating features of many such libraries, including PVM. However, it is very rigid and imposes requirements on how the programs must be structured that conflict with USS design.

c) Fast Messaging (FM - Karamcheti and Chien, 1994). This unsupported library provides a low-level IPC mechanism using shared memory routines which provides latency an order of magnitude lower than PVM. This would be the library of choice but even this could not overcome the other problems detailed below.

2. A process runs on one physical processor. There is no multi-threading support and this can only be achieved by using a large conditional statement in a monolithic program to select alternative execution paths. To port a multi-process system to a one process per processor architecture would have involved major changes and be grossly inadequate. The other alternative would be to have one system component running on each processor and treat the whole machine as one node. This would, of course, be absurdly inefficient since many processes, such as entities, are inactive for a large proportion of their life.
3. The Cray C++ compiler does not support exceptions which were used extensively in the prototype (section 5.2.5). Removing exception handling code from a program requires a total re-design and re-write.

4. Whilst it was possible to communicate from the T3D to the outside world through the Y-MP host using a “message-routing” process, the author was advised against trying. The host was so heavily used any such routing process would have to wait a long time to gain access to the CPU thus shattering any hope of reasonable real-time performance.

5.2.3 Sun SPARCcenter

The Sun SPARCcenter 1000E met all of the required criterion and was used to develop the prototype in parallel with the QNX version. SunOS v5.4 supports some of the POSIX standards which made porting relatively straight forward. However, this machine is used by many in the University as a compute server and therefore could not be used to evaluate system performance.

5.2.4 SGI RealityStation

A network of three SGIs arrived in the Department of Computer Science half way through the final year of this project. The most powerful of the machines was a RealityStation which is populated with 128 Mbytes of main memory and runs the IRIX OS (v5.3) which uses virtual memory. Unfortunately a suitable C++ compiler was not installed until a couple of months before submission, limiting work on this platform to a minimum.

5.2.5 Implementation Language

Development of the prototype started under QNX which supported ANSI C and C++ with exceptions and templates. In general, the code generated by a C++ compiler is as efficient as a C compiler and since object-oriented techniques lend themselves well to the task at hand, C++ was chosen as the implementation language. The availability
of templates eased development and exception handling helped produce an easier to understand implementation. Watcom C++ v9.52 was used under QNX and Sun Professional C++ v3.0.1 was used to initially develop the prototype under SunOS. Later, compatibility with GNU C++ v2.7.0 was tested as a precursor to the SGI port and to aid debugging (section 6.2.1).

Figure 5.4 Example network configurations of three USSs.

5.3 Networking

The actual organisation of USSs need bear no relation to the physical location of the nodes or their internetworking. Figure 5.4 shows three possible configurations of a USS, all of which are connected to the same backbone network. System Enterprise is constructed from three nodes interconnected by a dedicated network with one node acting as a gateway to the backbone. System Voyager only has one node whilst Defiant has two nodes but its local communications must share the bandwidth with all of the other traffic on the backbone. Whilst this last configuration is not efficient, it is functionally valid.

There is no required medium or protocol for interconnecting systems. In this example, however, all the nodes use Ethernet as their communications medium. It is
possible that the medium used within USS Enterprise could be totally different provided that an Ethernet link to the other systems was still maintained. This would be the situation in a multiprocessor system where each processor could correspond to a node.

Each OS has its own mechanism for sending messages to processes within its domain of control. On a single processor system this means sending messages between logical processes running on the same processor and may be implemented as either sharing or copying memory. This is also true in some multiprocessor systems where memory is shared, in others communications may use high-speed links between processors. In distributed systems the message may also be sent between physical machines over a high-speed LAN connection. The one criterion that links all these different domains is that the recipient is directly addressable by the operating system.

5.3.1 IPC Mechanisms

Most operating systems provide their own method of lightweight message-passing, e.g. QNX, but others rely on more heavyweight methods such as TCP/IP, e.g. IRIX. Under QNX, multiple machines may be networked together into one virtual machine and the system’s IPC mechanism works between processes on different nodes as if they were on the same physical machine. It can coexist in an Ethernet network with other protocols but cannot be used to communicate with systems that are not running QNX. In order to communicate with processes outside the native domain of control it is necessary to use a different delivery system, such as TCP/IP. This also means that a different addressing method must be used.

To localise the impact of these differences (and those of other OSs), a Process Management Layer (PML) is incorporated into each system component which sits in between the operating system and the component implementation (Figure 5.5). This process layer provides a set of services (presently just IPC) which are independent of the underlying operating system. Where more than one delivery system is available the layer chooses the right mechanism for the right job. How these decisions are made is platform and implementation specific. There is only one requirement, of
course: the message delivery must be **reliable**. The prototype supports QNX IPC, TCP/IP and the framework for supporting a shared memory IPC mechanism is present but not fully implemented. UNIX domain sockets (which are faster) were not used instead of TCP/IP because QNX does not support them and they would complicate system performance comparisons (section 6.4).

<table>
<thead>
<tr>
<th>Component Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Management Layer</td>
</tr>
<tr>
<td>Operating System</td>
</tr>
<tr>
<td>Hardware</td>
</tr>
</tbody>
</table>

**Figure 5.5 Position of the Process Management Layer within the system software.**

### 5.3.2 Addresses

Each process within the system has an address which is unique throughout all USSs. The address is made up of three components: the *system ID* (SID), the *node ID* (NID) within that system and the *process ID* (PID) within that node (Table 5.1).

Current sizes are signed 16 bit integers for both the SID and NID, with an unsigned 32 bit integer allocated for the PID. Valid SIDs and NIDs are positive integers - negative values are used during the process' initialisation phase. This provides a unique address for 32768 systems, each with up to 32768 nodes, each of which may have $2^{31}$ processes running on them. This is truly overkill for the prototype but offers a realistic address range when large-scale distribution is a goal.

<table>
<thead>
<tr>
<th>USS ID</th>
<th>USN ID</th>
<th>Process ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bits (signed)</td>
<td>16 bits (signed)</td>
<td>32 bits (unsigned)</td>
</tr>
</tbody>
</table>

**Table 5.1 Message address structure.**
When each system is defined in the systems’ configuration file (section 5.4), it is allocated a unique SID. Likewise, each USS definition contains a number of USN definitions which specify a NID that is unique within that system. The PID is different because the number used is unique within the given node. It is used to reference the process that the message is intended for (or sent by), but how it is used to locate the relevant process is implementation and thus node dependent. When using QNX IPC, messages are indeed addressed using the operating system’s process identifier, whereas an implementation using TCP/IP uses the socket number associated with the process. A shared memory implementation would use the address of the memory block holding the message queue.

5.3.3 Messages

All communication between the components of the USS use a number of pre-defined messages whose basic structure is shown in Table 5.2.

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Message ID</th>
<th>Transport ID</th>
<th>Length</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bytes</td>
<td>8 bytes</td>
<td>1 byte</td>
<td>1 byte</td>
<td>4 bytes</td>
<td>optional</td>
</tr>
</tbody>
</table>

† Aligned on a 2 byte boundary, i.e. requires one padding byte.

Table 5.2 Message header structure.

The address of the sender and the intended recipient are the first two fields in the message header. The recipient field is necessary because the message may be routed through one or more other processes before it arrives at its destination. The message ID number is used by all system components to determine whether to deal with the message and, if so, how to decode the data (if there is any). The desired method of transportation to the recipient is also recorded in the message.

The size of the associated message data is given in bytes. The interpretation of the data depends on the message ID. A list of the defined message types and their purpose is given in Table 5.4. Message IDs are often reused for slightly different purposes, the exact meaning depending on the receiver, e.g. entity, manager, etc. In addition, many messages share the same physical structure with regards to data.
contents (Table 5.3). For example, all messages that contain UML information (binary or ASCII) use the same structure: 7.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-</td>
<td>0</td>
<td>All information required is in the message header.</td>
</tr>
<tr>
<td>0</td>
<td>String</td>
<td>n</td>
<td>Used to send variable length textual information.</td>
</tr>
<tr>
<td>1</td>
<td>Notify</td>
<td>4</td>
<td>Holds reason for process termination.</td>
</tr>
<tr>
<td>2</td>
<td>Ping</td>
<td>28</td>
<td>Holds flag indicating whether receiver issued ping or is being pinged and timestamp information.</td>
</tr>
<tr>
<td>3</td>
<td>Profile</td>
<td>16+</td>
<td>Holds a variable length RP.</td>
</tr>
<tr>
<td>4</td>
<td>UPID</td>
<td>48</td>
<td>Room for both the name and UPID of a process.</td>
</tr>
<tr>
<td>5</td>
<td>UPID2</td>
<td>16</td>
<td>Contains just two unnamed UPIDs.</td>
</tr>
<tr>
<td>6</td>
<td>Status</td>
<td>4</td>
<td>Details the status of a previously requested service.</td>
</tr>
<tr>
<td>7</td>
<td>UML</td>
<td>24+</td>
<td>Holds either an ASCII UML definition or binary state data.</td>
</tr>
</tbody>
</table>

Table 5.3 Description of the nine physical message structures.

5.3.4 Hardware Differences

Sending messages between machines in a homogeneous environment requires no additional effort. However, in a heterogeneous network there are hardware architecture differences.

5.3.4.1 Byte Order

The byte ordering used in CPUs may be classed as either little-endian or big-endian. A little-endian CPU, such as those produced by Intel, places the least significant byte or a word first. Conversely, a big-endian CPU places the most significant byte first. The reasons behind the choice of one ordering over another will not be discussed here but recently some CPUs have been built such that the byte ordering used can be selected by setting a bit in one of the CPU’s registers, e.g. Motorola 88110 (Motorola, 1992).
<table>
<thead>
<tr>
<th>Message</th>
<th>Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET_UPID</td>
<td>4</td>
<td>Sent to the UM to obtain the sender's UPID.</td>
</tr>
<tr>
<td>SET_UPID</td>
<td>4</td>
<td>Sent by the UM, containing the recipient's UPID.</td>
</tr>
<tr>
<td>PING</td>
<td>2</td>
<td>Test connection/measure round-trip time to a given process.</td>
</tr>
<tr>
<td>NOTIFY</td>
<td>1</td>
<td>Inform the UM why this process is terminating.</td>
</tr>
<tr>
<td>RPROFILE_NOTIFICATION</td>
<td>3</td>
<td>Holds a process or node RProfile.</td>
</tr>
<tr>
<td>RPROFILE_REQUEST</td>
<td>3</td>
<td>Sent by a process wanting a process or node RProfile.</td>
</tr>
<tr>
<td>LOCATE_REQ</td>
<td>0</td>
<td>Ask the UM to locate a process based on the specified search criterion.</td>
</tr>
<tr>
<td>LOCATE_RESP</td>
<td>4</td>
<td>UPID of the located process returned by the UM.</td>
</tr>
<tr>
<td>STATUS</td>
<td>6</td>
<td>Success/reason for failure of the specified message.</td>
</tr>
<tr>
<td>ACTIVATE_UM</td>
<td>-1</td>
<td>Notify the MUM that this node is active.</td>
</tr>
<tr>
<td>DEACTIVATE_UM</td>
<td>-1</td>
<td>Notify the MUM that this node is disconnecting/tell slave node to terminate.</td>
</tr>
<tr>
<td>ACTIVATE_USS</td>
<td>-1</td>
<td>Notify the master USS that this system is active.</td>
</tr>
<tr>
<td>DEACTIVATE_USS</td>
<td>-1</td>
<td>Notify the master USS that this system is disconnecting/tell slave system to terminate.</td>
</tr>
<tr>
<td>TERMINATE</td>
<td>-1</td>
<td>Sent by UM to force termination of any given process.</td>
</tr>
<tr>
<td>CREATE_ENT</td>
<td>0</td>
<td>Execute the given process on the recipient UM's node.</td>
</tr>
<tr>
<td>CREATE_ENT_ACK</td>
<td>-1</td>
<td>Sent by SUM to MUM when an entity has been created.</td>
</tr>
<tr>
<td>DESTROY_ENT</td>
<td>0</td>
<td>Terminate the given process on the recipient UM's node.</td>
</tr>
<tr>
<td>DESTROY_ENT_ACK</td>
<td>-1</td>
<td>Sent by SUM to MUM when an entity has been destroyed.</td>
</tr>
<tr>
<td>UML</td>
<td>7</td>
<td>Holds valid UML code to be parsed by the recipient.</td>
</tr>
<tr>
<td>UML_INIT</td>
<td>-1</td>
<td>Request the sender's UML definition from the UM.</td>
</tr>
<tr>
<td>UML_INIT_DEF</td>
<td>7</td>
<td>New, complete UML definition sent by the UM.</td>
</tr>
<tr>
<td>UML_CONSTRUCT</td>
<td>7</td>
<td>Execute entity's Construct function/entity's initial state information.</td>
</tr>
<tr>
<td>UML_UPDATE</td>
<td>7</td>
<td>Execute entity's Update function/send state updates.</td>
</tr>
<tr>
<td>UML_DESTRUCT</td>
<td>-1</td>
<td>Execute entity's Destruct function.</td>
</tr>
<tr>
<td>UML_MONITOR</td>
<td>0</td>
<td>Manager's registration of interest in part of the UML definition.</td>
</tr>
<tr>
<td>UML_MONITOR_ACK</td>
<td>-1</td>
<td>Sent by the UM to confirm acceptance of a monitor request and inform entities of dependency.</td>
</tr>
<tr>
<td>UML_SYNC</td>
<td>0</td>
<td>Request current list of UML dependencies.</td>
</tr>
<tr>
<td>UML_UPDATE_NOTIFY</td>
<td>-1</td>
<td>Notify entities that they should update and managers that they should expect UML_UPDATE messages.</td>
</tr>
<tr>
<td>UML_UPDATE_COMPLETE</td>
<td>-1</td>
<td>Notify managers that all entities have updated.</td>
</tr>
<tr>
<td>MIGRATION-notification</td>
<td>5</td>
<td>Informs receiver that a migration has occurred - contains the process' old and new addresses.</td>
</tr>
<tr>
<td>MIGRATION_REQUEST</td>
<td>4</td>
<td>Sent by a RM to the MUM to request an entity migration.</td>
</tr>
<tr>
<td>MIGRATION_STATE_REQ</td>
<td>-1</td>
<td>Sent to an entity to obtain a complete copy of its state.</td>
</tr>
<tr>
<td>MIGRATION_STATE</td>
<td>7</td>
<td>Complete entity state sent from source to target entity.</td>
</tr>
<tr>
<td>MIGRATION_STATE_ACK</td>
<td>-1</td>
<td>Used to inform the MUM that state transfer was successful.</td>
</tr>
</tbody>
</table>

Table 5.4 Summary of message types and their use.
5.3.4.2 Floating-Point Representation

Another difference may be the representation of floating-point numbers: single-precision (32 bit), double-precision (64 bit) and extended precision (64 bit and upwards). This is less of a problem since most general-purpose CPUs conform to IEEE 854 (IEEE, 1987) although they may, of course, have a different byte order.

5.3.4.3 Memory Alignment

Some architectures also require certain data types to be aligned on given byte boundaries. For example, a 32 bit integer may have to start on a 4 byte boundary. If not required then often operations are performed more efficiently if aligned on these boundaries. In these cases the alignment is enforced by the compiler or provided as an option (Watcom, 1995).

5.3.4.4 Transfer Format

The External Data Representation (XDR) library of functions are used to represent data structures in a machine-independent form (Bloomer, 1992). This library is available on most machines running UNIX and can be used to encode dynamic data structures as well as just handling the primitive types. Due to this level of functionality it is also quite a bulky library with respect to both memory requirements and the API. Even the low-level code used by Snowdon (1995) produced a significant overhead.

Of the platforms available for use by the author, two used big-endian ordering, one used little-endian and all of them used the same single and double-precision floating-point formats. Since the UML data structure traversal routines had already been written and the number of messages types sent between machines was relatively low, it was decided to provide hand-coded byte-swapping routines. In addition, although XDR is a popular library, it may not be available on all systems which would cause problems porting USS.
The chosen format for sending messages was little-endian because the big-endian machines had more powerful CPUs and could better accommodate the overheads involved in encoding/decoding. The byte-swapping code was conditionally compiled into big-endian systems to minimise code size and maximise execution speed on little-endian machines. As the process layer receives messages, it encodes/decodes those that are destined for/received from other nodes.

5.3.5 Layer Implementation

Each process in a USS is both a provider and a consumer of services. A service is requested by sending a message to the provider which performs some processing and then possibly sends a result back to the consumer. Information flows between processes freely and it is possible for two processes to be each other's consumers and providers. The PML provides the nuts and bolts that can support this functionality and avoid deadlock.

5.3.5.1 Asynchronous

Synchronous message transmission is a convenient mechanism for issuing service requests but can leave the sender waiting for a response when it could be doing other work. In USS, therefore, all processes send a message and then continue immediately with other processing. Some time in the future they may receive a response to their original request which must be associated with it in some way. This may be explicit by including a reference in the response or implicitly because it could only have come from one message.

5.3.5.2 QNX

Messages are sent between QNX processes using a three stage procedure: Send-Receive-Reply. Figure 5.6 shows the sequence of these stages and what happens to the state of each process. After a message has been sent, the sending process blocks until it receives a reply from the message's recipient. Similarly, when a process enters the receive state it blocks until it is sent a message at which point it can do some
processing and then must issue a reply. It is possible to poll for a message but continuous use of this service will seriously degrade system performance. To minimise the time that the sender is blocked, a reply is issued immediately after receiving the message.

![Diagram showing Send-Receive-Reply procedure for sending messages under QNX.](image)

* Only when sending to another node.

Figure 5.6 Send-Receive-Reply procedure for sending messages under QNX.

Sending a message to another node in a QNX network requires the establishment of a virtual circuit between the sender and receiver. The identifier assigned to this circuit is then used when sending the message instead of the PID in the message address. After the reply has been received the virtual circuit is deleted. It would be more efficient to leave the virtual circuit in place and re-use it the next time - an operation supported by QNX. However, the burden placed on the operating system by the potentially large number of circuits could degrade system performance. The buffer used for sending messages within the operating system grows as needed but it is also possible to send multi-part messages which keeps the required buffer size at a minimum.

5.3.5.3 TCP/IP

Each process obtains a socket number which is used throughout its lifetime as the PID component of the UPID. Whilst the contents of the PID field in the message address
is enough to send a message under QNX, TCP/IP also requires a hostname to establish a socket connection. If the recipient is on the same node then the node's hostname can be obtained from the operating system. Any message destined for another node is sent through the UM which maintains a routing table for each node in the system. If it is the MUM then it also stores a route for its counterpart in each system. A table entry is composed of the SID, NID and hostname.

The sequence of events required to send a message using TCP/IP as implemented in the process layer is shown in Figure 5.7. TCP/IP requires a connection to be established before data transfer may commence. A similar phase is the creation of virtual circuits in QNX, but whereas QNX provides OS support for maintaining virtual circuits, it is up to the application to keep track of established socket connections. Each connection has to be periodically polled to check for incoming messages compared to issuing a single call to Receive(). Since this would introduce unwanted complexity and a considerable overhead in the prototype, socket connections are established and closed each time a message is sent.

5.3.5.4 Deadlock

A problem common to both of these implementations is that of deadlock. If process A should send a message to B at the same time as B sends a message to A then both will be blocked waiting for the other to receive the message. A solution is to split the layer into two processes. The first process holds all the components functionality and receives messages as per normal. When it wishes to send a message, it is passed to the second child process which actually sends it. Therefore only the child process ever becomes send-blocked leaving the parent process to accept incoming service requests and perform its usual work (Figure 5.8).

---

3 Stored in and administered by the PML.

4 These overheads would not be incurred if an unreliable datagram (connectionless) mechanism were used.
Figure 5.7 Message transmission sequence using TCP/IP.

The overheads of this solution can be minimised by using threads (lightweight processes) which share both code and data, with a separate stack (Milenkovic, 1992). Messages could then be passed from parent to child by exchanging memory pointers. Unfortunately threads have not been implemented on all of the chosen platforms. A beta version of a threads library was available in QNX but was found by the author to be unreliable and so this option was ruled out.

Figure 5.8 Structure of a logical process consisting of two physical processes.

The ability to create a child process using fork() is a common feature in UNIX-based systems. The child is, in effect, a duplicate of the parent process, sharing code but taking a separate copy of the data and stack. Although not strictly an IPC
mechanism, pipes are commonly used to send data between two processes on UNIX-based systems. Pipes fall under the jurisdiction of the filing system but that does not require them to occupy disk space and may reside totally in memory. Since both these features were available on the target platforms this method of implementation was chosen. To reduce the often considerable memory overheads that fork() produces through duplication of data and stack, the child process, once created, is replaced by a lightweight mailer. This program simply reads messages from the pipe and sends them to their intended destination.

5.3.5.5 Initialisation

The PML is the first software element to be initialised when a process is created. Its first task is to determine the UPID of the process it is executing in. If it is a UM then initialisation is temporarily paused whilst the configuration file is parsed and then restarted when the node’s SID and NID are known (section 5.4). The PID of the UM is the actual process identifier under QNX or a pre-defined port when using TCP/IP, i.e. 34000.

If the process is not a UM then it must locate its UM and send it a GET_UPID message. Location of the UM using TCP/IP is simply a case of connecting to the pre-defined port address. Under QNX the operating system’s name server is used to locate the process identifier of the UM using a pre-defined name.

Upon reception, the UM allocates a UPID and returns it in a SET_UPID message which is subsequently processed and thus completes the layer initialisation.

5.3.5.6 Multiple Mechanisms

The layer can be initialised to handle both QNX and TCP/IP IPC. If so, connections on each mechanism are polled for, in turn, until one is established. This is a CPU intensive procedure if done continuously, but it is commonplace for each component to poll once for any messages before continuing with the outstanding work (section 5.6.4). Consequently, multiple mechanisms may be handled with only slightly more overhead than just one.
When there is a choice of methods for communication, the mechanism specified in the message is used. If this is left undefined then the best choice is used - the prototype will use QNX IPC in preference of TCP/IP. It is, however, uncommon for a message's transportation ID to be left blank, since it is accepted practice to respond using the same method that the request was sent with.

5.3.6 Networking Summary

In order to simplify the transfer of messages between processes and facilitate porting to different platforms, each software component has a process management layer. The interface to this layer, the message format and message addressing are the same regardless of the OS. In a heterogeneous environment, a common binary format must be agreed upon to enable machines with different hardware architectures to communicate. In the present day, these differences are far fewer and a compromise was found quite easily. As messages are sent they are encoded into the common format (if necessary) and decoded upon receipt (if necessary). To avoid deadlock the PML requires two processes to be used per logical process: one with the component-specific functionality and a small mailer process used to send messages. The PML's first action during initialisation is to ascertain its UPID, either through a configuration file or by communication with the node's UM. Once initialised, the network of PMLs can handle message transmission between nodes using different IPC mechanisms.

5.4 Configuration Control

Some of the components in a USS need configuration information when they are created. This section presents a simple language that is used to help fulfil this task and is followed by an example of its application: system configuration.

5.4.1 Universal Configuration Language (UCL)

This minimalist language provides a way of structuring simple information in a hierarchical manner. UCL is used by those processes that need configuration information upon creation. The UCL parser constructs a small internal data structure
which may be read, manipulated by the process and then output again. Currently, this
information is stored in files which are read by each process but there is no reason
why this information could not be sent by the UM.

The basic building blocks of UCL are Components and Variables. A variable is given
a type of Real, Integer, String or Boolean and lists may have mixed types. Every
variable is required to have a value, but if this is not needed an empty string may be
specified (" "). A component can contain variables and zero or more other
components which form a hierarchy, of which there may be many in each file. Figure
5.9 shows a contrived example of a UCL description that contains one of each
possible construct.

Components are identified by a type name which is followed by an optional name that
can be used for reference purposes during parsing and when accessing the information
described therein.

```
Container containerName
(
    SubContainer componentName
    {
        aString "hello"
        aReal 1.0
        anInteger 2
        aBoolean FALSE
    }
    mixedList 1, 2.0, TRUE, "goodbye"
)
```

Figure 5.9 The basic elements of UCL.

UCL permits structuring of non-complex data in which ever way is most suitable for
the task at hand. In order for a UCL file to be recognised by different programs, the
type names of components and their structure must be made concrete. Such a process
was undertaken to provide a configuration file for USSs.

5.4.2 System Configuration

Figure 5.10 shows how UCL is used to describe the configuration of the USS
Enterprise shown in Figure 5.4. The node that has the MUM is indicated by the
presence of the MASTER variable which is used as a flag. Likewise, one of the systems in the configuration file must be designated as the master system, similar entries would be made for the two other systems (section 5.6.2). The SID of the first system description in the configuration file is 1, the second system is allocated a SID of 2, and so on.

```

USS Enterprise
{
    MASTER "" // Master system

USN Pentium
{
    HOST "haggis.psy", 2 // Host name and NID
    IPC "QNX" // Uses QNX IPC
    RM "resnode2.ucl" // Has a Resource Manager
    VISM "" // Has a VIS Manager
}

USN Server
{
    HOST "haggis.psy", 1 // Different node
    IPC "QNX"
    RM "resnode1.ucl"
    CONSOLE "" // Has a console attached
}

USN Gateway
{
    MASTER "" // Master node
    HOST "haggis.psy", 3
    IPC "QNX", "TCP/IP"
    RM "resnode3.ucl"
}
}
```

Figure 5.10 Example USS configuration file.

The HOST variable specifies the hostname of the node and its NID. It is necessary to describe the location of the systems/nodes in some meaningful way and the hostname’s format is dependent upon the protocol used to interconnect systems. In the prototype, TCP/IP is used and the hostname is therefore given in Domain Name Server (DNS) form. The IPC mechanisms supported by the node are also listed, two of the nodes only use QNX IPC whilst the Gateway node also supports TCP/IP. Since this node is the link to the other systems it is also designated as the master.

The remaining entries correspond to the managers that run on each node. All nodes have a resource manager entry which takes a file containing its initialisation
parameters. The only special manager in this system is VIS which runs on the machine with the CIG. However, one node does have the system console attached for the administrator's use.

5.5 A UML Interpreter

Before examining each system component it is important to understand how the UML interpreter works because it has had considerable influence on their implementation. There are four stages to interpreting a UML description:

1. Lexical analysis.
2. Syntactical and grammatical verification.
3. Construction of the interpreter's internal data structure.
4. Semantic validation of that data structure.

The first stages were accomplished by using the lex and yacc tools (Levine et al., 1992). The product of these tools was combined with a series of C++ classes to form a UML interpreter library which could be linked into any program requiring that ability. Manipulation of the interpreter is possible through the library's API.

There are two phases when building the data structure: first of all the data definition is parsed and then all instruction code is compiled into an intermediate byte-code. This section describes the general structure of this library and outlines the processes of interpretation.

5.5.1 Overall Structure

At the highest level, the structure of UML may be conceptualised as a list of universe and entity definitions. Each of these definitions may be linked to one another by inheritance or they may just be peers with a common ancestor. Every universe definition is itself a hierarchy of other components: elements, constants, properties, etc. Each entity is derived from one of the universe definitions and contains a number of scope levels with functions, variables, etc., forming yet another tree structure.
Each component of UML has been implemented as a C++ class which are all derived from a common base class called UMLComponent (Figure 5.11). The base class holds data structures that are essential to each component class.

The UML object acts as the top-level interface to the interpreter and the data structure representing the UML description. The other objects correspond exactly to the UML constructs described in section 4.4.

### 5.5.2 Interpreting the Data Definition

When a component description is encountered, its position within the data structure is first determined. At the top-level the parser may encounter any component - all but the universe and entity definitions use the dot notation. If the component is a universe then it is added to the UML object whilst an entity description results in its definition being added to the object. All other components require their corresponding stub declaration to be located and their description modified. Nested component definitions may be added to the relevant component data structure directly.

After all UML statements have been successfully parsed, the data structure undergoes a validation process. Universes may be derived from other universes and elements from other elements. If a component is derived from another, then that parent component is sought for and a link is made between the two components. An entity

---

5 Instancing the UML class creates the interpreter and therefore there is only one UML object per process.
description is always derived from a universe and a similar link is made between the entity and the host universe. Failure to locate a parent component is a fatal error and parsing ceases. When an element is specified as the type of a property then a similar search is made and a link established.

The search for a given component starts in the current scope and, if it is not found, progresses outwards. If the host universe/element has a parent then this is also thoroughly searched and its ancestors, if necessary, until a result is obtained. Failure to locate the host component results in an interpreter error.

The way that the data structure is modified is affected by the current mode of operation, i.e. insert, replace or delete (section 4.4.3.1.9). By using these mode directives as stream modifiers it is possible to modify the UML definition in the course of usual interpretation rather than through the library API. At the completion of the interpretation, a single unified data structure has been built which holds all the UML descriptions passed to the interpreter, regardless of original physical location.

5.5.3 Instancing

At this stage no space has actually been allocated for any data. First an instance of the relevant portion of the data structure must be created. This could be the whole structure, e.g. instancing a universe, or just one element or built-in type, e.g. instancing a property.

When a compiler, e.g. C++, builds a map of any given data structure, each component is allocated a chunk of memory contiguous to the previous allocation. Storing all instance data together in such a container is a sensible thing to do since the data structure is static and will not change at run-time. The same technique is used in many interpreters for the same reason. However, this technique will not work with UML since the structure is dynamic and may be altered at any time.

One possible solution would be to use the same contiguous allocation of memory but store pointers to the relevant chunks in the UML data structure. In other words, each component would know whereabouts its instance data is in the container. When a
change is made, e.g. a new component added, then a new container would be allocated and the existing components’ data copied into it, inserting the new data in the process. A complementary technique could be used for deletion. Obviously this solution would require an amount of container memory greater in size (for the insertion case) than the existing instance data to be allocated before the process could commence. If a complex component was being altered then this could potentially be very large and at the very least result in a considerable amount of time spent copying data from one container to another.

A better approach would be to scrap the idea of storing all instance data in one place and instead store it individually. Whilst this requires a larger overhead in both memory and processing time to locate the instance data, it does mean that modifications to the UML structure do not require large memory allocations or copying. All instance data is kept associated with their definition as indicated in Figure 5.12. In this example there is one instance of the Outer element and two of Inner, one for the innerInst property and the other for the local function variable. Whilst the property instance will exist as long as that property is part of the universe definition, the variable instance will be created when the function is entered and destroyed when it has completed.
The process of instancing may be directly applied to a universe, property or function variable. In fact, for all intents and purposes, a variable and a property are functionally equivalent. Instancing a universe actually results in each of the universe's
properties being instanced. If the universe has no properties then it has no state. Each property has an _instance list_ which maintains a record of each instance of that property and they are distinguished through the use of an _instance identifier_ (IID). An IID is a signed 32 bit integer, thus supporting 2147483648 instances during the life time of the universe⁶. IIDs are allocated to each component in the order in which they are instanced. If the property’s type is an element then that element’s properties are also instanced and so on until the bottom of the component tree is reached. For example, outerInst would have an IID of 1, innerInst would be 2, number would be 3 and local is 4. When a list is instanced each entry is assigned a unique IID.

Consider the case when the definition is altered by the insertion of a new property - _vector_ - as shown in Figure 5.13. After the data structure has been modified and validated, instancing merely requires allocating IIDs and memory for 3 real numbers and adding links to them in the instance list. The rest of the data structure has not been modified in any way and the original contents of the instance data for Outer have been preserved. Similarly, if innerInst was deleted then vector would be unaffected.

In the absence of an initialiser for any given property, the default values assigned are: zero for real and integers, false for booleans, and strings are empty. This assignment is also repeated within any element that a property may instance.

State indexing (section 4.4.3.2.5) was not implemented but would require adding an extra dimension to the instance list of each property that used the feature.

---

⁶ Negative values are used for internal purposes.
5.5.4 Component Dependencies

A key feature of UML is the ability to establish a dependency on a particular part of the definition (section 4.5.4.6). The functionality to handle dependencies is defined in the class from which all components are derived - UMLComponent. Figure 5.14 shows its structure and that of a skeleton dependency. Just as each of the UML components are derived from UMLComponent, so each application uses UMLDependency as a basis for the information it needs to store for each
dependency. An example of this specialisation is given in section 5.6.3.1 which describes how the UM uses this data structure.

![Diagram of UMLComponent structure with dependency](image)

**Figure 5.14 UMLComponent structure with dependency.**

Dependencies are made on different components with respect to the dependent’s needs. This mechanism is used internally to detect when functions which access a given component may need to be re-interpreted. It is also used by managers to keep track of changes in the values of properties, among other things.

Each dependency may be given the state of *active* (default) or *inactive*. A monitor may deactivate a dependency to avoid the overhead of removing it and then re-establishing it later on. A count of the active dependencies is maintained in the component. After a new dependency has been added or an old one removed, the monitor typically builds a dependency list. This list is usually used to process each interest in turn and perform some (often recursive) operation. If we had already registered interest in `innerInst` and now we became interested in `outerInst`, it could, at least, result in a duplication of effort and at worst, end in processing `innerInst` twice. There are therefore two ways of building a dependency list. A full list includes all components with dependencies, whereas a partial list does not include any component which is inherited from another component in the UML hierarchy with an active dependency (i.e. below an active dependency).

### 5.5.5 Interpreting Instruction Code

The part of the interpreter that deals with instruction code was given a low implementation priority due to time constraints. The author felt that the exact
features of the programming language should be carefully considered. Also, further exploration of existing byte-code engines would be required to derive a sufficiently efficient interpreter. Furthermore, implementation was not necessary to prove the viability of the system architecture. Consequently the instruction code interpreter has not been implemented. However, some of the implementation issues are presented here for consideration by the reader.

There are two common methods for interpreting code. The first performs syntax and grammatical analysis each time, effectively interpreting the ASCII statements in their raw form. The second compiles those same statements into an intermediate code which is then executed by an automata. The overhead of parsing the original statements at execution time is large in relation to the execution of a set of pre-compiled instructions. It is true that less memory is required for the storage of intermediate code than the original ASCII text, but this must also be kept in some form if future re-interpretation becomes necessary.

For these reasons UML instruction code is first compiled into an intermediate byte-code which is stored in the data structure and may be executed by a byte-code engine at any time. During the compilation various components will be referenced, either in variable declarations, i.e. elements, or expressions modifying state, e.g. properties. If these components do not exist or there are any syntactical or grammatical faults then an error is flagged. Accesses to instance data refer directly to the data itself and therefore do not require any data structure traversal. This means that any additions to the element will not require the code to be automatically re-interpreted. Deletions, however, can cause havoc.

The removal of an element or any component within an element that is depended on by code requires the re-interpretation of that code. How the functionality of the code has been affected by the change in structure cannot be ascertained without some form of artificial intelligence. Even then, comprehension of how this code segment fits into the larger picture is far more complex and would require human intervention. Consider the expression \( a = b * c \). If component \( b \) is deleted from the definition
we are left with $a = c$. This may still be valid or it may be wrong, only within the context of the rest of the code can a decision be made. Faced with the possibility of receiving dozens - if not hundreds - of requests for help from the UML interpreter, it seems sensible to at least provide some tool to aid the process. The best that can be offered is an arbitrary component expression eliminator that would remove references to the deleted component(s) whilst still retaining syntactic and grammatical correctness. The resultant code could be offered to the modifier as a potential solution and then rejected/accepted as required.

The code may, of course, be changed at any time through the API. The origin of these changes may be from a human or another program within the system. Thankfully this is a straight-forward task to complete since it is identical to the process undertaken when parsing the original code as detailed above.

### 5.5.6 Interpreter Embedding

As development of the simulation progresses, some definitions and associated code will be reused over and over again. The Read and Write routines declared in section 4.4.3.1.3 for managing visual information could potentially be used in every entity. Translation of such UML code into the native IL would be sensible for performance reasons. Access to the interpreter’s data structures is possible via the library API and the execution of native machine code (rather than UML) will be transparent to the application. The IL routines are usually placed in a library and linked in with each application that needs them. The ability for an entity to migrate to other nodes need not be affected if:

1. The destination node has its own native version of these routines.

2. The original UML code is at hand and may be used when native code is not available.

Permitting the interpreter and ILs to interact provides a powerful basis with which simulations may be developed. UML code may be used for lightweight tasks and
rapid prototyping of more complex functions which, when finalised, may be coded in the IL.

5.5.7 Persistence

Since the complete definition is either represented by a data structure (in the case of the data definition) or by the original text (in the case of the instruction code), it is possible to output any part of a UML definition at any time. This ability is very useful when changes have been made at run-time and the original definition is now incorrect.

To migrate an entity requires the transfer of its essence from one place, i.e. the UML definition and its current state. Fortunately the definition can always be reconstructed from the state so it is only necessary to send the latter. The same process is also required in order to save the current state of an entity to backing storage so that it may be reloaded in the future.

The state is the sum of all the instance data and packaging it, by necessity, involves the manipulation of binary data. If this package will be sent to another node then, in a heterogeneous network, it may not share the same architecture. Following the decision made in section 5.3.4, three routines are defined in the IL for every component: size, pack and unpack. The size routine traverses the given definition and estimates the size of each of its components, producing a grand total at the end. This figure is used to allocate a buffer into which pack stores the data by once again traversing the data structure. Each component's instance data is preceded by a small header providing vital information to aid its extraction by unpack. When packing or unpacking the data on a little-endian machine no binary conversion is necessary, overheads are only incurred on big-endian systems.

5.6 Universe Manager

There are three main stages to the execution of the UM. First of all the UM's node must be initialised, at which point it is ready to join the network of other nodes
comprising the system. Once this connection has been established, it enters an event loop which processes service requests that are sent to it and also generated internally.

5.6.1 Node Initialisation

As the first process to start, the UM is responsible for configuring its node and if it is the MUM, organise the system. After the PML has paused its initialisation, the first action taken is to process the configuration file. Its local node and the master node information is located, as well as location information for the other systems if it is the MUM. As each node/system is processed the UM builds a routing table for those systems that are connected via TCP/IP. Now that the SID, NID and PID are known, the PML completes the initialisation of the IPC mechanisms.

At this point the execution paths differ for SUMs and MUMs. If it is running on the master node then the location of the UML definition is verified and interpreted. All SUMs locate their MUM and send it an ACTIVATE_UM message. Afterwards, all UMs create any managers that are configured for their node, starting with the RM and then the specialised managers. The creation of a console is initiated by the administrator and may be performed at any time.

5.6.2 System Initialisation

After manager creation, system initialisation is completed. The MUM waits for activation messages from each SUM which it acknowledges. This acknowledgement changes the node’s state to alive. When all nodes are alive the system itself is deemed to be alive.

In the prototype a multi-level hierarchical system organisation is not supported, rather a simple master/slave structure has been adopted. In the same way that there is one master node in a system, there is one master system (MUSS) and zero or more slave systems. Any communications that must be sent to other systems are sent directly to

---

7 The filename is passed as a command line parameter.
the MUSS which routes them to all the other systems. Therefore, after the MUM has initialised its system, the address of the MUSS is sought and stored explicitly for future use.

### 5.6.2.1 Load Balancing

Rather than obtain a full RP from each RM, the prototype uses a simple CPU rating in the current load-balancing algorithm to determine on which node the declared entities in the universe definition should execute. Each time an entity is created, the optimum distribution of processes between nodes is recalculated and the entity is allocated to the node that has the largest difference to its optimum load. Table 5.5 shows the debugging output from the load-balancing algorithm. The figures inside brackets represent the ideal number of entities for each node if another entity is created, whilst those outside are the current distribution of entities.

<table>
<thead>
<tr>
<th>Current Entities</th>
<th>Entities on Server</th>
<th>Entities on Gateway</th>
<th>Entities on Pentium</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 (0.204)</td>
<td>0 (0.224)</td>
<td>1 (0.572)</td>
</tr>
<tr>
<td>2</td>
<td>0 (0.408)</td>
<td>1 (0.447)</td>
<td>1 (1.145)</td>
</tr>
<tr>
<td>3</td>
<td>1 (0.612)</td>
<td>1 (0.671)</td>
<td>1 (1.718)</td>
</tr>
<tr>
<td>4</td>
<td>1 (0.816)</td>
<td>1 (0.894)</td>
<td>2 (2.290)</td>
</tr>
<tr>
<td>5</td>
<td>1 (1.020)</td>
<td>1 (1.118)</td>
<td>3 (2.863)</td>
</tr>
<tr>
<td>6</td>
<td>1 (1.224)</td>
<td>1 (1.341)</td>
<td>4 (3.435)</td>
</tr>
<tr>
<td>7</td>
<td>1 (1.427)</td>
<td>2 (1.565)</td>
<td>4 (4.008)</td>
</tr>
<tr>
<td>8</td>
<td>2 (1.631)</td>
<td>2 (1.788)</td>
<td>4 (4.580)</td>
</tr>
<tr>
<td>9</td>
<td>2 (1.835)</td>
<td>2 (2.012)</td>
<td>5 (5.153)</td>
</tr>
<tr>
<td>10</td>
<td>2 (2.040)</td>
<td>2 (2.235)</td>
<td>6 (5.725)</td>
</tr>
<tr>
<td>11</td>
<td>2 (2.243)</td>
<td>3 (2.459)</td>
<td>6 (6.298)</td>
</tr>
</tbody>
</table>

Loading was based on CPU ratings of 260, 285 & 730 respectively. Figures in brackets represent the new optimum load for each node to 3 sig. fig.

**Table 5.5 Sample entity distribution over three nodes.**

The first row of the table shows that the first entity was allocated to Pentium. The fastest and least loaded node is always chosen for the target when the next entity is created, which in this case means Gateway with a predicted loading of 0.224. This result is confirmed by the second row in the table which also shows that the next
entity will be allocated to Server and so on. When there are ten entities the home for the new entity is Gateway. This is because Pentium is overloaded by 0.275, Server can only handle 0.04 more entities and Gateway has room for 0.235 entities. A total of the entities active on each node is kept at all times.

Currently there is no way of associating an RP with a specific entity so each entity is allocated an initial default profile.

5.6.2.2 Entity Creation

Originally it was planned for the MUM to extract the relevant portion of the UML definition and send it to the destination node’s UM. However, if an entity should migrate to a node that does not have the entity’s definition it must be sent prior to the migration, thus increasing the time taken to complete this operation. Therefore each UM has a complete copy of the UML definition. Since the instruction code part of UML has not been implemented, the entity’s functionality is written in the implementation language and executed in place of interpreted code (section 5.8.1). Normally there would be one generic entity process with a built-in UML interpreter to start, but because functionality may differ between entities, a specific executable must be identified. The prototype takes the name of the entity and translates this into the name of an executable that exists within the search path of each UM. A CREATE_ENT message is then sent by the MUM to the target node indicating the name of the executable. On receipt of this message the process is started, indication of success is sent back in a CREATE_ENT_ACK message and entity execution continues as usual (section 5.8). Of course, if the entity is executed locally then the process is merely started and the MUM moves onto the next entity.

---

8 This path can be modified using the ENTPATH variable in the node’s configuration section.
5.6.3 Managing Processes

Information about each process running on the node is held by the local UM in a *process list*. The structure of a process entry is shown in Figure 5.15. Every process is allocated one of six types: RM, ENT, MAN (special manager), MUM, SUM and CON (console). There are three states that processes progress through during their lifetime. After execution has started, but before the process has been allocated a UPID, it is allocated the state of *genesis*. When the initial handshaking is over and the process is ready to satisfy service requests it is said to be *alive*. During the termination process, after it has ceased to function in the simulation *per se*, the process is said to be *dead*. When termination is complete the entry and its dependent structures are removed from the list.

Any given UM holds information about every entity and manager running on its node; if it is the MUM, information on any SUMs is also held; if it is a SUM, its MUM's details are stored. Treating parent and child UMs as processes running on its node simplifies certain procedures that the UM must perform, e.g. dependency management (described below).

5.6.3.1 Component Monitoring

When a manager wishes to monitor a given UML component, its absolute name (using dot notation) is sent within a *UML_MONITOR* message. After verifying that this component actually exists the manager's information is found within the process list, a new dependency is created and added to the process' *dependency pool*. The pool is essentially a fixed size array which provides fast entry lookup. As dependencies are removed, gaps appear but these are filled as new dependencies are added.

The *UMDependency* information is derived from *UMLDependency* as described in section 5.5.4 and adds a pointer back to the owner's process entry (Figure 5.15). This organisation permits any *process* to locate all of the components it is dependent on and any *component* to determine which processes are dependent on it. Although
the framework is here to support dependencies on any component, only monitoring of properties is currently implemented.

The monitor ID returned to the manager is actually the component’s index in the process’ dependency pool. The UM must now inform all relevant processes that a new dependency has been established using a UML_MONITOR_ACK message. As each entity is processed a new dependency is also added to their pool; its index provides the monitor ID to be used in communications with this entity. Both the MUM and the SUMs are also informed using the original message sent by the manager. Each add a dependency to the sending UM’s process entry and inform the sender of the monitor ID to be used in further transactions regarding this component. Without keeping a process entry for parent/child UMs, this procedure would be far more complex than necessary. If an entity is created after all dependencies have been established then a current list is sent as a stream of separate messages.

Figure 5.15 Structure of the information held for each process.

5.6.3.2 Component Updates

When an entity sends a state update to the UM, its process information is retrieved and the dependency pool entry described by the monitor ID in the message is extracted. From this point a list of those processes dependent on this state is
available. Each dependent's unique monitor ID is extracted from their pool and placed into the message before it is forwarded to it by the UM. Figure 5.16 presents an example where the component state has an ID of 1 when it is sent to the UM, but has the values of 4 and 2 when forwarded to the two interested managers.

No extra space is required to store each monitor ID because it is the index into the dependency pool. The only computational overhead incurred is a simple pool lookup as each dependent is processed. Constraint functions were not implemented because they rely upon the UML instruction code interpreter which was also not implemented.

![Figure 5.16 A state update uses a different monitor ID when sent to each dependent.](image)

5.6.4 Processing Service Requests

Two features common to all component implementations are the event loop and the action queue. When an internal function wishes to perform more than one action, e.g. send a message, or can/needs to spread its work over a period of time, then it enqueues a token representing the pending action (with parameters) in the action queue. The event loop checks if there are any external service requests which it processes first to maintain responsiveness. If there is not a message waiting then it dequeues the next action and performs it. If there are not any actions to perform then the process simply blocks until a service request arrives. It is not uncommon for one action to enqueue another during its execution.
One action that must be performed in the initial stages of a UM's lifetime is waiting for all entities and managers to complete initialisation, the specifics of which are described in the following sections. When all entities and managers are alive the simulation loop is entered which sends a \texttt{UML\_UPDATE\_NOTIFICATION} message to each entity and manager. After all entities have updated, a \texttt{UML\_UPDATE\_COMPLETE} message is sent to all managers and after they have updated the next notification message is sent and so on. The state update process triggered as each entity completes its update has already been described and the following sections discuss this and the managers actions in more detail. Other service requests/actions that are intermingled with this sequence are location requests, entity executions, synchronisation requests, etc.

If an entity should terminate abnormally and a destruct message has not been issued then the UM will do so on behalf of the late entity. This ensures that the simulation does not become full of zombie entities whose state copies are still being maintained by managers.

### 5.6.5 Entity Migration

In order for entity migration to be implemented it is necessary to have some basis upon which to make decisions about node loading. This was done through the use of CPU consumption alone. However, without a fixed time frame to relate these measurements to, a CPU usage is useless. This fixed period would normally be provided by the scheduler and equate to one simulation step, but since a full scheduler was not implemented a simple step duration threshold was used for the migration test presented in section 6.5.4. The intention is to keep the simulation step duration below the threshold through use of migration. Each step, the RM totals the amount of CPU used by the entities and if it exceeds the threshold the migration mechanism is be triggered. In this prototype the MUM does not decide when migrations should take place but relies upon each RM to volunteer entities.

When the mechanism is invoked, the entity with the largest CPU usage is identified and its UPID sent to the MUM in a \texttt{MIGRATION\_REQUEST} message. The requests, of which there may be more than one generated by different nodes each step, are
enqueued and then processed at the end of the current simulation step. The source node of each request is excluded from selection in the load-balancing algorithm and the optimum distribution is calculated as if the system has one less node. Once a suitable target node has been found, an entry is added to the MUM's migration list which details those entities in the process of migrating and their current status; specifically, their name, source node, target node and source UPID.

The next stage is to create a copy of the entity on the target node using the normal creation procedure. Once this has been done, a MIGRATION_STATE_REQUEST is sent to the original entity which packs up its entire state and returns it to the MUM in a MIGRATION_STATE message. This is then forwarded by the MUM to the newly created entity which unpacks it and, upon success, sends a MIGRATION_STATE_ACK back to the MUM. Finally the original entity is terminated by sending a DESTROY_ENT message to the entity's UM, MIGRATION_NOTIFICATION messages are sent to all managers (including SUMs) and the entity's migration list entry is removed. The notification message simply contains the old and new UPIDs for the entity and enables the managers to update their internal data structures accordingly. The UM on the source node uses this information to re-route any messages that are sent by processes unaware of the migration. After forwarding the message, the UM sends the originator a migration notification message so that this does not happen again.

Currently any error that occurs during the entity migration, e.g. failure to create the target entity, is treated as fatal and the migration request is ignored.

5.6.6 System Interaction

The multi-system functionality that has been implemented is limited to group initialisation, termination and the transmission of changes in the UML definition. Inter-system user functionality has not been implemented, e.g. shadow entities, because it is hard to demonstrate in a thesis and was therefore given a low priority.
5.6.7 System Termination

A system termination is invoked from the MUM by first sending termination messages to each SUM. The MUM and SUMs then send termination messages to their local managers and destruct messages to all their entities. Once all processes on a slave node have terminated the slave informs the MUM that the node is shutting down with a DEACTIVATE UM message. Finally, when all the local processes on the MUM and its slaves have terminated, the MUM ends execution.

5.7 Resource Manager

The implementation of the RM is quite simple because there is no scheduler. Subsequently the RM keeps track of the resource utilisation for its node and makes rudimentary judgements about its loading.

![Diagram of Resource Consumption](image)

**Figure 5.17** Resource consumption representation.  
a) class hierarchy; b) Resource Profile structure; c) Resource History structure.

5.7.1 Resource Consumption

Each resource has been implemented as a class derived from one base class (Figure 5.17a). An RP is composed of these different types: a list of CPU consumption (for multiprocessor systems), a list of memory usage (used in those systems with special memory architectures) and a record of space used on different storage devices. The
totals of each of these are also stored and is supplemented by the network usage (Figure 5.17b). The prototype actually only makes use of the CPU information.

The RM maintains a resource history for each process (Figure 5.17c) which contains the process’ last RP, its current profile and a prediction of future resource requirements (currently unused).

```
RM
{
  CPU Pentium_90MHz
  {
    Manufacturer "Intel"
    Integer 0.849  // BYTEMark integer index
    FloatingPoint 0.881 // BYTEMark floating index
    ICache 8  // Kb
    DCache 16 // Kb
    IntThreshold 90.0 // %
    FPThreshold 90.0 // %
  }
  MEMORY Main
  {
    Size 24576 // Kb
    Access 70 // ns
    Threshold 80.0 // %
  }
  STORAGE Primary
  {
    Size 524288 // Kb
    Access 12 // ms
    Threshold 95.0 // %
  }
  NETWORK Ethernet
  {
    Bandwidth 6.0 // Mbps (Effective)
    Threshold 40.0 // %
  }
}
```

*Figure 5.18 Example node resource configuration used by a RM.*

### 5.7.2 Initialisation

Each node’s resources are detailed in a file (written in UCL) which is passed as a command-line parameter to the RM when it is started. Figure 5.18 shows an example configuration of the Pentium node which details the CPU type (an Intel Pentium/90), the total system memory, backing storage and network link bandwidth.
For the migration tests a CYCLE variable was used at the top level to specify the threshold duration of the simulation step in milliseconds.

### 5.7.3 Services

After the configuration information has been processed, the main event loop is entered. Initial work usually consists of processing the RPs sent by each entity as it is created and keeping the UMs informed of the current loading. During the period before the system goes live it is not possible for an entity to overload a node since it has been carefully allocated by the MUM. However, as soon as the entity starts executing it may provide modifications to its RP based on its expected resource consumption. Since a full scheduler was not implemented, this detailed information was not needed. For the same reasons, the RM does not keep the MUM informed of node loading. Instead the RM tells the MUM when load balancing is necessary.

Since this prototype instills the progression of the simulation with the MUM rather than the scheduler in the RM, an UPDATE_NOTIFICATION message is sent to the RM at the end of each simulation step. This is the RM’s cue for assessing CPU usage and when this is complete an UPDATE_COMPLETE message is sent back to the UM. The simplest information on a process’ execution time under UNIX-based operating systems is provided in the form of user and system times. These represent the total CPU used by the process when executing system calls (system) and when executing application code (user). The current RM adds these figures together to get a CPU usage figure for each process. By monitoring the previous usage the process’ consumption for the last simulation step can be ascertained.

When the migration mechanism is being used, the total of these times is used to decide whether the entity with the highest CPU usage should be migrated. Currently the CPU thresholds are not used, instead the step duration variable (CYCLE) is consulted for the desired time. If the total CPU time used by all entities exceeds this time then a migration request is sent to the MUM. The RM is informed of a successful migration with a MIGRATION_NOTIFICATION and subsequently removes the entity from its calculations.
5.8 Entity Library

The core entity functionality has been placed in a library which works on two levels. Once initialised, its event loop enables it to correctly interact with other processes in the system and, through the use of a function call-back mechanism, can be tailored for a specific purpose. The source code of an example entity can be found in Appendix B.

5.8.1 Initialisation

Following PML initialisation, the call-back table is reset and specific call-backs may be registered. An entity handles all the UML messages in addition to those dealing with RPs, location responses and monitor acknowledgements. The first message processed by the entity is its RP which can then be modified. After locating the RM, the RP is sent to it and a request is made for the entity's UML definition.

Normally entity behaviour would be exhibited through execution of UML code, but since the instruction code interpreter has not been implemented, functions written in the IL must be used. Typically the only call-back used is that for the UML_INIT_DEF message which is used to send the entity its definition. At this point the entity's UML Construct, Update and Destruct function declarations are located and defined as embedded IL routines as opposed to UML code. When these functions are executed by the UML interpreter, the IL routine is called. Access to the state information is obtained through the UML API.

5.8.2 Service Requests

The first external events received by the entity are indications of monitored components in the form of UML_MONITOR_ACK messages. Unlike the UM, the only information that the entity need keep track of for each dependency is the monitor ID contained in the message. This dependency list is rebuilt each time a new monitor notification is received.
Upon receipt of a construct message the UML interpreter is instructed to construct the entity's state. On completion an instance ID is returned which is used in all further accesses to the state information. The Construct function is then executed, thus initialising the state and is followed by the enqueuing of the action to send initial state updates to the UM. Receipt of an update results in the same execution-action sequence. The current component dependency list is used to determine which state updates to send. Asides from executing the destruct function, no further action is taken when an entity destructs. The PML, by default, informs the UM of the process termination and whether it did so naturally or not.

When a MIGRATION\_STATE\_REQ message is received by the entity, it packages up its complete state and sends it back to the UM in a MIGRATION\_STATE message. Upon termination the entity destructs as normal. When the target node is sent the state message it instances its definition and unpacks the state into the newly created instance. The construct function is not called and a UML\_CONSTRUCT message is not sent to the UM. From this point on, however, the target entity takes over all processing from the original and operates normally, issuing state updates as necessary.

5.9 Manager Library

The manager functionality has been structured in a similar manner to that of an entity. On its own, the library will interact correctly with the other process' in the system but does not perform any special manager-specific tasks. This higher-level functionality is added through the call-back mechanism. Appendix B contains an example of this library's use.

5.9.1 Initialisation

Following the usual process initialisation, the manager is sent the complete UML definition and (through a call-back) registers interest in the specific components it uses. Each manager maintains a monitor list with an entry for each component it is monitoring (Figure 5.19). An entry consists of a pointer to the relevant portion of the UML data structure for that component and the monitor ID used in communications
with the UM. The three other essential call-backs are those for `UML_CONSTRUCT`, `UML_UPDATE` and `UML_DESTRUCT`. It is within these functions that the heart of the special manager's functionality is embodied. An example of their use is given in section 5.10.

At the lowest level the manager keeps an entity list. An entry is added to this list on receipt of a construct message, modified by an update message and removed when an entity destructs. An entry exists for each monitored component held by each entity. When a construct message is received for a component, that part of the UML data structure is instanced and the contents of the message unpacked into the state instance. The instance ID is stored in the entity list entry along with the entity's UPID and a pointer to the relevant entry in the monitor list. This enables the location of all state information related to a specific entity with minimal redundancy.

5.9.2 Simulation Loop

Each simulation step starts with the reception of a `UML_UPDATE_NOTIFY` which can be used via a call-back to perform preliminary work for each update. When an update message is received the monitor entry is located using the message's monitor
ID. Then the component's state is located by searching the entity list using the entity's UPID and the monitor entry as keys. The new state is then unpacked into the instance and the update call-back executed if present. When a \texttt{UML\_UPDATE\_COMPLETE} is sent by the UM the simulation step has concluded and the manager may perform (via call-back) any final calculations before the next step. The return of a status message to the UM indicates that the manager has completed its work. This start/stop message system is necessary because an entity will not send an update unless that component has been modified. Therefore there is no way for a manager to determine whether all updates it should be sent, have been sent. A destruct message results in the deletion of that entity's component instance and then the removal of the relevant entry from the entity list.

When a \texttt{MIGRATION\_NOTIFICATION} is received, the manager locates the old entity's entry in the entity list and replaces the UPID stored therein with the new address in the message. No other action is needed.

5.10 Visual Manager

The prototype VIS implementation does not interface to a CIG since it was not deemed necessary in order to demonstrate the effectiveness of the USS architecture. In fact, it is not used when evaluating the system's performance in the next chapter, but it is presented here as an example of a special manager implementation.

The code used to explore the viability of real-time VE displays was available for use (section 5.1.1) but was not utilised for two reasons. Firstly, there is no way to satisfactorily demonstrate such a feature in a thesis. Secondly, graphics and API speed is totally CIG dependent and would only confuse any analysis of the manager's performance. Therefore, everything apart from the actual calls to the CIG's API was implemented.
5.10.1 Initialisation

Following the standard manager initialisation the prototype VIS registers interest in the Base.models.visual and Base.models.position properties (section 5.10). At this point the CIG would also be initialised and initial parameters set, e.g. viewpoint position, etc.

VIS registers call-backs for all construct, update, destruct and update-complete messages. As each entity constructs, VIS receives a stream of construct messages which are acted upon by the call-back function. This is responsible for creating the initial visual representation of the entity in the CIG database.

5.10.2 Simulation Loop

As updates are sent to VIS, the update call-back is executed which is used to move the entity's representation and if necessary, modify it. On receipt of an update complete notification the new scene is rendered and the manager has finished its work for the current step. Destruct messages result in the removal of the representation from the CIG database.

5.10.3 Entity Enhancement

The extra functionality needed by any entity wishing to manipulate its visual representation is provided in the form of a library. Whereas this could be provided as importable UML code, it is currently IL code which is linked into the ENT executable. The Read and Write function definitions (section 4.4.3.1.3) are supplemented with internal routines which may be used to manipulate the Visual element data structure.

Therefore, an entity's construct call-back function will build the visual representation, either from file or by code. The update call-back modifies the state as necessary and the destruct call-back closes the library.
5.10.4 VIS Summary

The current VIS implementation is very basic but it performs the essential operations required of it. Since all the complex operations are hidden in the manager library, the developer can concentrate on what the manager should be doing and implement it with the minimal coding.

5.11 Console

The console implementation is a hybrid of a manager and an entity in that it receives most messages in order that it may keep track of the system’s status. A command-line interface provides the opportunity to display this information and issue simple commands. An entity creation, destruction or migration request may be sent to the UM from the console, as can UML code. The console keeps an up-to-date copy of the complete universe definition although it does not maintain any instance data. The current functionality is quite limited and was used for testing purposes only.

5.12 Further Improvements

At this stage, it is apparent that a number of enhancements can be made to the prototype.

5.12.1 Configuration

The configuration information required by child processes, e.g. the RM, is currently passed to them as a filename in their execution parameters. This has two disadvantages: firstly, it introduces a dependency on backing storage and, secondly, it increases the process initialisation time. If this information was passed to them by the UM, both these problems could be overcome. This would not require changing the current configuration file format and could be sent in its native ASCII format.
5.12.2 Multi-part Messages

Presently the PML relies on the operating system to break large messages into smaller packets for transmission. This ability is not supported by all IPC mechanisms and therefore the addition of PML controlled multi-part messages would be advantageous. This would also reduce the amount of buffer space required to send a message and permit the construction of messages whose total length is not known when the first part is sent.

5.12.3 State Encoding

With the ability to gradually build a message, the estimation of state size prior to encoding may be removed. Instead the state may be encoded directly into a multi-part message thus substantially reducing the time taken to send state updates.

Alternatively, memory could be allocated during packing, building a linked list which is then traversed when copying the state into the fixed size message buffer. This, at least, removes the need to estimate size initially.

5.12.4 Persistence

The current implementation assumes that a simulation will run to completion before the system terminates. Therefore no provision is made for state persistence such that a simulation may be saved and reloaded at a later date. In order to realise this, an entity could be sent a TERMINATE message before destruction which would be its queue to save its state to backing storage. Upon restarting a simulation the entities would be created as before (but possibly not on the same node) and during construction their state loaded from backing storage. Managers can rebuild their internal data structures from the events that would take place upon restarting the simulation, e.g. entity creation, initial state transmissions, etc. It may be necessary, however, that those structures unique to each manager are also saved for use when the manager re-initialises.
5.12.5 Message Elimination

The three messages UML_CONSTRUCT, UML_UPDATE and UML_DESTRUCT sent to an entity should be replaced by a UML message which simply executes the Construct, Update or Destruct function respectively. The resulting state updates generated by these calls would be returned in a standard state message which would include the name of the function that generated the data. All such remote code executions would operate in the same manner. The current shortcut was taken because the UML interpreter was not complete.

5.12.6 Entity Synchronisation

Synchronising an entity involves the transmission of multiple messages detailing individual monitor notifications. In this special case it would be preferable to send a single message containing all notifications, thus reducing the UM's overhead for this operation.

5.12.7 Function Access

At present, anybody may execute a function in an entity if it knows its name. This could be changed by providing a function hiding mechanism, e.g. a PRIVATE keyword to be used in the function declaration (not definition). Any attempt by a remote process to execute a private function would result in an appropriate exception generated by the interpreter.

This technique could be generalised by ensuring that any private function cannot be executed outside its scope. In Figure 5.20, unprotected may call protected since it is in the same scope but control may only call Inner.unprotected.
5.13 Summary

Before the details of the prototype USS were given, the implementation of a simple worst-case scheduler was described which has been used to enforce a constant-rate display. The experience gained by the author during this implementation and its subsequent use indicated that implementing scheduler functionality at the application level was not practical. The USS prototype implementation presented therefore did not make use of the scheduling aspects detailed in the design.

A layer of abstraction is introduced in the form of the PML in order to shield the USS processes from each operating system’s idiosyncrasies. Presently it is only used to provide a messaging service between both local and remote processes. The simple configuration language was then described and a typical example of its use presented in the form of the USS configuration file. The structure of the UML interpreter was described in terms of the data definition and instruction code sections. This included a detailed explanation of the complex data structure used to hold the model description and its instance data.

Each of the required system processes were dealt with in turn, describing the implementation of the basic operations they perform and services they provide. Special attention was given to the important data structures and how they are utilised at run-time. Most of the UM’s functionality was implemented including an elementary migration and load-balancing mechanism (using a minimal RM). The bi-directional data structure used by the UM permits the location of all components that
A given process is dependent upon and *vice versa*. The operations involving state transmissions and monitor IDs were described in conjunction with details of the relevant parts of the manager and entity implementations. The core entity and special manager functionality is provided as libraries which are specialised through the use of UML code and call-backs. An example of this is given with reference to the Visual Manager.

The chapter concluded with a few improvements that may be made to the current implementation. These functional changes will be supplemented by performance enhancing suggestions in the next chapter.
Chapter 6

Prototype Evaluation

"The mark of a truly civilised human being is the ability to read a column of numbers and then weep.”

Bertrand Russell

Evaluating a system implementation can be undertaken at two levels: component and system. A component analysis examines each system component in an isolated manner whilst a system analysis is holistic and operates at a higher-level, considering more functional problems. Indeed there is a fine balance to be struck between being too specific which produces results that do not mean anything useful, and being so general that there is no content to the results. The component level provides useful information that can aid development and testing but suffers from a lack of relevance when a system task is considered. At the system level the whole system is asked to perform some useful task and evaluation of its performance can be used to judge its overall effectiveness.

These methods are not mutually exclusive, in fact understanding system performance is difficult if the effects that the individual system components have are not fully understood. However, a component’s behaviour will often change when used in conjunction with other components within a system, e.g. its performance may be reduced when it has to bid for CPU time with other processes. This chapter, therefore, deals with the system as a whole (an approach advocated by Checkland, 1994) but with a detailed look at the two major components of most (if not all) system processes: the UML interpreter and the PML.

225
6.1 System Analysis

Ideally one would like to compare the performance of this prototype with that of other solutions for distributed VE systems. However, the only evaluation of a VE system that the author has found is for AVIARY (section 2.3.8), with the exception of a predictive performance chart for DIS (Figure 2.3). Even if figures were readily available, the problems that must be faced when comparing systems are similar to those encountered when comparing CIGs. Each manufacturer presents a list of figures which detail the CIG's performance of certain tasks, e.g. rendering a 10 by 10 grid of polygons, in relatively useful units, e.g. polygons per second. Unfortunately, information essential for comparison of the CIG's results with another CIG is often not presented, e.g. were the polygons lit? Clipped? Textured? With which texturing technique? etc.

The obvious course of action would be to derive a set of benchmarks that may be used to provide a fair basis for comparison of systems. But even this has problems, for example some CIGs are optimised for triangles whilst others can handle polygons with any number of vertices. Undoubtedly, any test using triangles will give any CIG optimised for this type a better rating than the other CIGs. Conversely, a benchmark that tested polygon throughput with varying numbers of vertices cannot be run on a triangle-only system without extra application processing to split the polygons into triangles, thus defeating the objective. There are many other examples of architectural differences that confound comparison.

The architectures of distributed VE systems are even more diverse than that of CIGs and presents a challenge when designing benchmarks. In the same way that a geometrical model can produce different performance ratings on different CIG's, VE system performance is very application specific. This may be a reason why figures are not available for existing systems - even the evaluation of AVIARY is based around an Air Traffic Control application. No attempt will be made in this chapter to derive a set of useful benchmarks since this is a subject suitable for a thesis in itself, a more basic approach will be used instead.
This thesis has already established that the user is the final judge of the system’s effectiveness and that certain criteria must be met to provide a usable interface (section 3.3). Although this prototype was not built to test these measures, it is possible to extract the most important feature of any such system which is the ability to progress the simulation as fast as possible. A suitable metric is simulation steps per second and is used as the absolute measure of this prototype’s performance.

6.2 Testing Methodology

All of the benchmarks used in this chapter were run under similar conditions. Normal operating system processes were reduced to a working minimum in order to maximise the available memory and minimise interference with the USS processes. No users were permitted access to the machines during testing and normal Internet services were suspended. Disk accesses only occurred at the beginning of a test and at the end when results were logged. Even then, only local storage was used, which was especially important in the case of the SGI where normal user directories are held remotely and accessed using the Network Filing System (NFS). This fact combined with the presence of virtual memory can drastically affect performance.

This section documents the relevant characteristics of the machines used to test the prototype and highlights a number of issues that affected system performance.

6.2.1 CPU Performance

Table 6.1 shows the relative performance of several Intel CPUs present in IBM PCs and the MIPS processor used in SGI’s RealityStation. The performance ratings are, of course, dependent upon the efficiency of the compiler and its ability to generate optimised code. The Watcom C++ compiler was used on the Intel-based platforms whilst GNU C++ (G++) was used on the SGI machine. The native C++ compiler was not used because it did not support exceptions but unfortunately the GNU compiler had a number of faults that presented problems. Firstly, the code optimiser could not be invoked if the source code used exceptions, subsequently the SGI’s performance was severely undermined. The figures shown inside the brackets are those of the
native C++ compiler with optimisation and those outside the brackets represent the results obtained using the GNU compiler without optimisation\(^1\). Secondly, the implementation of C++ templates is less than efficient with the current release of G++ and requires the instantiation of each template within each and every module it is used (GNU, 1995). Consequently, the executable sizes produced are much larger than necessary which in turn has implications for the amount of paging required during execution.

<table>
<thead>
<tr>
<th>Gateway i486 50 MHz</th>
<th>Server i486 66 MHz</th>
<th>Pentium Pentium 90 MHz</th>
<th>Reality MIPS 4400 200 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>0.256645</td>
<td>0.330643</td>
<td>0.849287</td>
</tr>
<tr>
<td>Floating-point</td>
<td>0.173911</td>
<td>0.211827</td>
<td>0.881350</td>
</tr>
<tr>
<td>Memory (available / total)</td>
<td>14 / 20 Mb</td>
<td>10 / 16 Mb</td>
<td>17 / 24 Mb</td>
</tr>
<tr>
<td>Bus</td>
<td>16 bit (ISA)</td>
<td>16 bit (ISA)</td>
<td>32/16 bit (PCI/ISA)</td>
</tr>
<tr>
<td>Bus Speed</td>
<td>50 MHz</td>
<td>33 MHz</td>
<td>30 MHz</td>
</tr>
<tr>
<td>Bus Bandwidth</td>
<td>95 Mbytes/sec</td>
<td>63 Mbytes/sec</td>
<td>114/57 Mbytes/sec</td>
</tr>
<tr>
<td>Disk</td>
<td>1 Gb</td>
<td>1 Gb</td>
<td>750 Mb</td>
</tr>
</tbody>
</table>

The CPU speeds were obtained using BYTE Magazine's BYTEmark benchmark program. A rating of 1.0 is equivalent to a DELL Pentium 90 MHz PC running DOS. The figures given include the machine's multi-tasking operating system overheads. Figures in brackets represent the native compiler's performance on the SGI.

ISA - Industry Standard Architecture
PCI - Peripheral Connect Interface

Table 6.1 Resource ratings for each test platform.

6.2.1.1 QNX

The total memory available on each platform running QNX is shown in Table 6.1 as well as the actual amount that may be used by non-system software. Since QNX does not provide any virtual memory this limits the number of system processes that may

\(^1\) The benchmark code did not contain exceptions and thus could be optimised, resulting in performance only slightly worse than that produced by the native compiler. However, these results would not be indicative of the prototype's performance and hence the unoptimised figures are given.
run at one time. The absolute maximum number of executables running simultaneously is 250 which allows for a maximum of 50 virtual circuits² (QNX, 1995).

Each of the three QNX machines (Pentium, Server and Gateway) are interconnected by a private Ethernet LAN using the same make of Ethernet card and the same Industry Standard Architecture (ISA) bus. Gateway has a second interface card installed which is connected to the university’s backbone network.

6.2.1.1.1 Scheduling

There are three different scheduling methods that any given process may be assigned to under QNX: First In First Out (FIFO), round-robin and adaptive. When using FIFO scheduling a process executes until either it voluntarily relinquishes control (blocks) or is preempted by a higher-priority process. FIFO is only of real use to ensure mutual exclusion when two processes are sharing a resource. Round-robin is like FIFO except that each process may also stop executing if it reaches the end of its timeslice (100 ms). Adaptive scheduling uses decaying priorities for those processes that consume their timeslice and priority boosts for those processes that are starved of CPU for one second or more.

The last scheduling policy is commonly used in systems where interactive and compute-intensive processes share the same machine, however it does make performance evaluation of a network of interacting processes difficult. All processes within the USS application were therefore placed in a round-robin scheduler at the same priority. This causes considerable starvation of the normal interactive processes (using the adaptive scheduler) but not to USS processes such as the Console.

² More virtual circuits may be supported by reducing the number of executables. There will be no such limits in the next major release of the operating system (v4.3).
6.2.1.2 IRIX

The limits imposed by IRIX on the number of executables, etc., were not reached by the prototype system and therefore did not interfere with the system testing. There were, however, two other issues which presented problems.

6.2.1.2.1 Scheduling

IRIX also supports different scheduling methods: real-time, deadline, timesharing, gang batch and batch. Normal interactive processes run in the timesharing queue while the deadline scheduler enables time constraints to be applied to a process - although its effectiveness is uncertain when invoked on a single processor system. Processes assigned to the real-time queue are guaranteed better performance than those in the timesharing and batch queues. Unfortunately, unlike QNX, only the super-user may promote processes to queues above the timesharing level. Due to present departmental policy, access to the test platform at this level was not granted to the author and therefore all USS processes were subject to adaptive scheduling in the timesharing queue.

6.2.1.2.2 Virtual Memory

Performance can also be compromised through the paging to and from disk that is undertaken when using virtual memory. Ideally all of each process’ code and data would remain in memory, as with QNX. This is possible under IRIX but super-user access is again required and therefore all results obtained under IRIX are confounded by irregular and uncontrollable paging activity.

6.2.2 Computation and Communication

The main emphasis on the resources consumed by the prototype has been split between computation and communications. Sending/receiving messages requires CPU and therefore any computation rating is affected by communications. This relationship is examined when analysing the PML and its results can be used to aid estimation of specific service overheads, e.g. registering interest in a particular UML
component. The other side of the equation in this example is the time it takes to manipulate the interpreter’s data structure. Such information is provided by the section on UML which also deals with the resource that has, to date, been overlooked by system evaluations: memory.

<table>
<thead>
<tr>
<th>System Component</th>
<th>OS</th>
<th>Code Size</th>
<th>Data Size</th>
<th>Total Size</th>
<th>Required Libraries</th>
<th>Executable Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>UML Library</td>
<td>IRIX</td>
<td>247,984</td>
<td>59,904</td>
<td>307,888</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>QNX</td>
<td>78,191</td>
<td>77,940</td>
<td>156,131</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>UCL Library</td>
<td>IRIX</td>
<td>46,768</td>
<td>38,768</td>
<td>85,536</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>QNX</td>
<td>13,478</td>
<td>37,803</td>
<td>51,281</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PML Library</td>
<td>IRIX</td>
<td>52,270</td>
<td>26,032</td>
<td>78,302</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>QNX</td>
<td>22,385</td>
<td>26,509</td>
<td>48,894</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Entity Library</td>
<td>IRIX</td>
<td>32,496</td>
<td>4,016</td>
<td>36,512</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>QNX</td>
<td>9,601</td>
<td>3,144</td>
<td>12,745</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Manager Library</td>
<td>IRIX</td>
<td>33,840</td>
<td>3,600</td>
<td>37,440</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>QNX</td>
<td>36,830</td>
<td>6,838</td>
<td>43,668</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RProfile Library</td>
<td>IRIX</td>
<td>47,776</td>
<td>4,806</td>
<td>52,582</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>QNX</td>
<td>10,436</td>
<td>2,296</td>
<td>12,732</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Message Library</td>
<td>IRIX</td>
<td>9,232</td>
<td>2,336</td>
<td>11,568</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>QNX</td>
<td>1,822</td>
<td>144</td>
<td>1,966</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Mailer</td>
<td>IRIX</td>
<td>6,416</td>
<td>960</td>
<td>7,376</td>
<td>PML, Message</td>
<td>430,984</td>
</tr>
<tr>
<td></td>
<td>QNX</td>
<td>3,756</td>
<td>1,019</td>
<td>4,775</td>
<td>-</td>
<td>119,704</td>
</tr>
<tr>
<td>RM</td>
<td>IRIX</td>
<td>32,976</td>
<td>3,568</td>
<td>36,544</td>
<td>PML, Message, RProfile, UCL</td>
<td>541,576</td>
</tr>
<tr>
<td></td>
<td>QNX</td>
<td>24,971</td>
<td>5,275</td>
<td>30,246</td>
<td>-</td>
<td>163,053</td>
</tr>
<tr>
<td>UM</td>
<td>IRIX</td>
<td>265,104</td>
<td>27,200</td>
<td>292,304</td>
<td>PML, Message, RProfile, UCL</td>
<td>1,180,552</td>
</tr>
<tr>
<td></td>
<td>QNX</td>
<td>84,916</td>
<td>18,044</td>
<td>102,960</td>
<td>-</td>
<td>355,173</td>
</tr>
<tr>
<td>Benchmark Manager</td>
<td>IRIX</td>
<td>8,448</td>
<td>1,152</td>
<td>9,600</td>
<td>PML, Message, UML, Manager, RProfile</td>
<td>787,336</td>
</tr>
<tr>
<td></td>
<td>QNX</td>
<td>4,153</td>
<td>1,192</td>
<td>5,345</td>
<td>-</td>
<td>269,725</td>
</tr>
<tr>
<td>Benchmark Entity</td>
<td>IRIX</td>
<td>10,656</td>
<td>1,424</td>
<td>12,080</td>
<td>PML, Message, UML</td>
<td>787,336</td>
</tr>
<tr>
<td></td>
<td>QNX</td>
<td>4,601</td>
<td>1,482</td>
<td>6,083</td>
<td>RProfile, Entity</td>
<td>265,689</td>
</tr>
</tbody>
</table>

†All sizes are given in bytes.

Table 6.2 Minimum memory usage of USS components.

6.2.3 Memory

Table 6.2 gives a breakdown of the sizes of each USS process in terms of code size, initialised and uninitialised data, and total executable size. A list of the required USS libraries associated with each process is also given in the table. Under IRIX the majority of an executable’s size comes from other general-purpose libraries provided
with the compiler, e.g. system call library, maths library, C++ iostreams library, etc. These libraries are much smaller under QNX, for example the total amount of USS code used in the mailer is 48,894 + 1,966 + 4,775 = 55,635 bytes, meaning that the system libraries account for 64,069 bytes. This is a worst case scenario since the amount of space used by these libraries will remain roughly the same for the larger executables. The figures for the data given above do not include the memory that the process may allocate during execution for dynamic data structures, etc. The large difference in IRIX and QNX code sizes is in part due to not using code optimisation and also the different CPU instruction sets.

A simple way to reduce the amount of memory required by each process is to make use of shared libraries. Such a mechanism places commonly used routines into a special library which is loaded once into main memory. A stub library is also compiled and is linked into the executable in place of the larger original. When a function in the stub library is called, the equivalent routine in the shared library is executed. In theory, the unique overheads incurred by each USS process may be reduced substantially since most libraries are used many times, e.g. the PML library and the UML interpreter.

The implementation of shared libraries under QNX is based upon the mechanism used by UNIX System V Release 3.2 which has an explicit interface for importing and exporting data into and from the shared library (QNX, 1994). Whereas managing data and code separately is a perfectly adequate approach for C-based applications, the technique cannot be extended to the object-oriented paradigm which deals with code and data together. Specifically, problems occur with C++ when virtual functions, static initialisers or exceptions are used. Therefore it was not possible to exploit shared libraries with the QNX implementation.

This is not entirely true because the C system libraries are shared which, for example, means that every mailer only needs 55,635 bytes of memory, not 119,704 bytes. With USS shared libraries this could be reduced to 4,775 bytes or lower. Similar improvements would be seen for the other processes.
IRIX does support shared libraries but because the system was not available until very late in the project their potential was not explored. It is important to note, however, that the use of shared libraries would not only reduce the amount of memory required by a USS process, but should also reduce the amount of paging under IRIX. A large commonly used shared library has a greater chance of staying in physical memory than several large executables, each with their own copies.

6.2.4 Instrumentation

All of the data in this chapter was collected by instrumenting key execution paths with timing code. A suitable number of iterations were executed for each test case, e.g. message size, and the averaged data is used in the charts. The amount of iterations and the type of instrumentation used was determined by taking clock resolution and (erratic) operating system overheads into consideration. Under QNX, the system clock has a resolution of 0.1 ms and the SGI has microsecond accuracy. For events that completed faster or close to the clock resolution, such as some of the UML interpreter operations, the total time taken to perform all iterations was measured, adjusted for loop overheads and then averaged. For longer operations, such as the simulation execution stages, each iteration was measured individually and then averaged. In all cases the impact that the profiling code had on the measurements was taken into consideration.

6.3 UML

Quantifying the resources consumed by the interpreter permits the designer to gauge the impact their simulation will have on the system. To this end a series of simple benchmarks were used to establish resource consumption on each of the test platforms. Since these tests were compute bound, the same pattern of relative platform performance is evident in each test. Therefore, quite often only the figures from one platform will be used in the graphical illustrations of the results. A full table of the results upon which this subsection is based, along with the simple UML code used, may be found in Appendix C.
6.3.1 Code Size

The one disadvantage of sending UML code between processes is that a complex description can take an appreciable amount of space. Table 6.3 shows two possible techniques for reducing the size of UML code for transmission. Compression is a method that can be applied to any kind of data, but those algorithms that work on repeating patterns, such as the Free Software Foundation’s GZIP, work well with textual data. Simply applying compression to the original UML description can result in approximately a 60% reduction in size. Another technique which can be used is that of tokenisation - it is unlikely that this would be used just before transmission but during the initial interpretation. Tokenisation simply replaces the language’s ASCII keywords by single-byte tokens and reduces the whitespace used to a minimum. If the tokenised form is compressed the relative effects are less because tokenisation is a simple form of compression. However, compared with just compressing the original, the code can be reduced to around 35% of its original size.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Original</th>
<th>Compressed</th>
<th>Tokenised</th>
<th>Tokenised &amp; Compressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>ts.uml</td>
<td>5910</td>
<td>2279 (38.5%)</td>
<td>4950 (83.8%)</td>
<td>2141 (36.2%)</td>
</tr>
<tr>
<td>base.uml</td>
<td>1131</td>
<td>469 (41.5%)</td>
<td>825 (73.0%)</td>
<td>392 (34.7%)</td>
</tr>
</tbody>
</table>

The Free Software Foundation’s GZIP was used to compress the ASCII UML files. All sizes are give in bytes; percentages represent the compressed size in relation to the original size.

Table 6.3 Effects of techniques to reduce code size.

Of course, compression comes at the cost of increased computational requirements. There is a minimum code size that compression will have a beneficial effect upon and, even then, the computation time sacrificed to achieve this makes the usefulness of such an operation dubious. Apart from the initial definition sent to processes upon creation, it is predicted that most UML code sent will be quite small, e.g. function invocations, minor code redefinitions, etc. It would seem practical, therefore, to restrict the use of compression to large messages and then only with hardware support.
6.3.2 Primitive Types

Table 6.4 shows how much memory each primitive type uses on the test platforms. Although this is dependent on the machine's architecture and compiler rather than the operating system, the latter classification is used for convenience in this and some subsequent tables. The boolean type is much larger than it could be but alignment on a four-byte boundary simplified the state encoding/decoding routines and thus improved performance. The difference in the memory used by a string is due to the different C++ String class implementations provided with each compiler.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Usage (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>QNX</td>
</tr>
<tr>
<td>Integer</td>
<td>Integral number</td>
<td>4</td>
</tr>
<tr>
<td>Real</td>
<td>Floating-point number</td>
<td>4</td>
</tr>
<tr>
<td>Boolean</td>
<td>Boolean</td>
<td>4</td>
</tr>
<tr>
<td>String</td>
<td>Character string</td>
<td>$16 + len$</td>
</tr>
</tbody>
</table>

Table 6.4 Memory consumption of the four primitive UML types.

6.3.3 Component Sizes

Figure 6.1 presents some simple formulae which may be used to estimate the memory usage of a UML component, from a literal to all of the modeled universes. Table 6.5 provides some approximate sizes of each component. The basic overheads are those that are needed merely to declare the relevant component; this will include the requirements of the base class if it is a derived component. Those overheads that are dependent on the definition being interpreted, e.g. adding a property to an element, are specified on an individual basis. These figures do not represent the variable amounts of dynamic memory that may be used in the basic overheads, e.g. the storage of strings representing names, etc. Therefore the total obtained from the use of this table will always be less than the actual memory usage. In addition the values given are dependent upon the hardware architecture (section 5.3.4) and the C++ compiler used. For example, there is no standard method of implementation to handle virtual functions in derived classes. The remainder of this section presents brief textual notes on each of the main components.
Component = Basic + (number of dependents * (overhead + Dependency))
Dependency = Basic + [size of derivatives]

Literal = Basic + [length of string]
Constant = Basic + Component + (number of literals * (overhead * Literal))
Function = Basic + Component + [return type]

Element = Basic + Component + (number of elements * (overhead + Element)) +
(number of properties * (overhead + Property)) +
(number of functions * (overhead + Function)) +
(number of constants * (overhead + Constant)) +
(number of converters * (overhead + Converter))

Property = Basic + Component + (number of instances * (overhead + Instance))
Instance = Basic + (size of list * overhead per list entry)

Universe = Basic + Component + (number of elements * (overhead + Element)) +
(number of properties * (overhead + Property)) +
(number of functions * (overhead + Function)) +
(number of constants * (overhead + Constant)) +
(number of converters * (overhead + Converter))

Entity = Basic + Component + (number of constants * (overhead + Constant))
(UML = Basic + Component + (number of universes * (overhead + Universe))
(number of entities * (overhead + Entity))

N.B. Square brackets [] represent optional portions of a component.

Figure 6.1 Basic relationships between UML components and their memory usage.

6.3.3.1 Component

All UML components are derived from the one base class, UMLComponent. An
overhead is incurred for each dependency associated with a component in addition to
the actual dependency structure. The skeleton dependency provided with the UML
library only holds a single flag but, as shown in section 5.6.3, the extensions added by
each application must be incorporated into this figure.
6.3.3.2 Literal

A literal stores either an integer, a floating-point number, a boolean flag or a character string. Dynamic memory is only allocated when storing a string, the amount being dependent upon its length.

6.3.3.3 Constant

A constant may be a list in which case an overhead is present for each list element, plus the actual size of each Literal.

6.3.3.4 Function

The memory used by a function is substantially increased when a return type has been declared.

6.3.3.5 Element

As one of the container components, an element can use greatly varying amounts of memory. Essentially, each component contained within the element requires information to be stored about its location.

6.3.3.6 Property

Whilst a property declaration is only held once in memory, the bulk of the memory consumption attributed to it is used when instancing it.

6.3.3.7 Instance

An instance is a list of pointers to the actual instance data. Therefore, each list entry incurs an overhead in addition to the actual data size which can vary from 4 bytes for most primitives, to any amount for an element.
6.3.3.8 Universe

The type of overheads detailed in Figure 6.1 are the same as those for an element except that the minimum size is slightly smaller.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Usage (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>QNX</td>
</tr>
<tr>
<td>Component</td>
<td>Basic</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>Overhead per Dependency</td>
<td>12</td>
</tr>
<tr>
<td>Dependency</td>
<td>Basic</td>
<td>4</td>
</tr>
<tr>
<td>Literal</td>
<td>Basic</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>Overhead for a string of \textit{len} characters</td>
<td>\textit{len}</td>
</tr>
<tr>
<td>Constant</td>
<td>Basic + Component</td>
<td>76</td>
</tr>
<tr>
<td>Function</td>
<td>Basic + Component</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>Optional return type</td>
<td>44</td>
</tr>
<tr>
<td>Element</td>
<td>Basic + Component</td>
<td>140</td>
</tr>
<tr>
<td></td>
<td>Overhead per Element</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Overhead per Property</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Overhead per Function</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Overhead per Constant</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Overhead per Converter</td>
<td>12</td>
</tr>
<tr>
<td>Property</td>
<td>Basic + Component</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td>Overhead per Instance</td>
<td>12</td>
</tr>
<tr>
<td>Instance</td>
<td>Basic</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>Overhead per list entry</td>
<td>12</td>
</tr>
<tr>
<td>Universe</td>
<td>Basic + Component</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>Overhead per Element</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Overhead per Property</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Overhead per Function</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Overhead per Constant</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Overhead per Converter</td>
<td>12</td>
</tr>
<tr>
<td>Entity</td>
<td>Basic + Component</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>Overhead per Constant</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Overhead per Function</td>
<td>12</td>
</tr>
<tr>
<td>UML</td>
<td>Basic + Component</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>Overhead per Universe</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Overhead per Entity</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 6.5 Approximate memory usage for UML components.

6.3.3.9 Entity

The overheads for an entity are relatively small currently because instruction code is not stored, only constants and functions.
6.3.3.10 UML

A single instance of the UML interpreter holds references to all of the universes defined and the entities that exist within them. Following the data structure tree from this point enables us to determine the amount of memory used by the interpreter.

6.3.3.11 Example

Table 6.6 shows a small segment of a UML data definition. Using the data for QNX presented above, it shows how much memory would be used to represent the definition's structure within the interpreter and hold a single instance of element Triangle. Each component within an element automatically generates a 12 byte administration overhead in addition to the structure needed to hold that component's information. When creating an instance of a property, a 24 byte administration overhead is incurred and a further 12 bytes for every entry in a list. In the case of the coord array, this means that 60 bytes are used to manage 12 bytes of actual instance data, whereas 60 bytes are used to manage 216 bytes of the instance data for vertexList.

<table>
<thead>
<tr>
<th>UML Definition</th>
<th>Representation Size</th>
<th>vertexList Instance Size</th>
<th>coord Instance Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELEMENT Triangle</td>
<td>140</td>
<td>24 + (3 * (12 + 4))</td>
<td>72</td>
</tr>
<tr>
<td>{ ELEMENT Vertex</td>
<td>12 + 140</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{ PROPERTY coord : REAL[3] ;</td>
<td>12 + 88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>} PROPERTY vertexList : Vertex[3] ;</td>
<td>12 + 88</td>
<td>24 + (3 * (12 + 72))</td>
<td>72</td>
</tr>
<tr>
<td>}</td>
<td>492</td>
<td>276</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.6 Example of how much memory is allocated to represent a UML definition and hold its instance data under QNX.

Any instance of a property with a primitive type will have a disproportionate amount of memory used to manage the instance versus storing the instance data. The reasons for this complexity have already been discussed (section 5.5.3). Although a special arrangement might be made for properties of a primitive type, this would make the interpreter more complex and probably increase execution time.
6.3.4 Interpretation

Figure 6.2 shows the relative time taken to perform the three basic interpretation operations (insert, replace and delete) for three primitive components on each of the test platforms. In the case of the element and entity components, the definitions used in the test had no contents so that the measurements would be representative of each component. The property was given an arbitrary primitive type (integer) for the same reasons. Similar measurements were performed for functions and constants but give results very close to that of the property because the same amount of memory is currently used to represent them internally. Complete details may be found in Table 6.7.

<table>
<thead>
<tr>
<th>Component</th>
<th>Action</th>
<th>Pentium (ms)</th>
<th>Server (ms)</th>
<th>Gateway (ms)</th>
<th>Reality (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELEMENT</td>
<td>Insert</td>
<td>0.308</td>
<td>1.160</td>
<td>1.032</td>
<td>0.284</td>
</tr>
<tr>
<td></td>
<td>Replace</td>
<td>0.376</td>
<td>1.397</td>
<td>1.192</td>
<td>0.314</td>
</tr>
<tr>
<td></td>
<td>Delete</td>
<td>0.313</td>
<td>1.117</td>
<td>0.946</td>
<td>0.244</td>
</tr>
<tr>
<td>CONSTANT</td>
<td>Insert</td>
<td>0.339</td>
<td>1.202</td>
<td>1.094</td>
<td>0.303</td>
</tr>
<tr>
<td></td>
<td>Replace</td>
<td>0.381</td>
<td>3.911</td>
<td>1.202</td>
<td>0.318</td>
</tr>
<tr>
<td></td>
<td>Delete</td>
<td>0.297</td>
<td>1.060</td>
<td>0.917</td>
<td>0.230</td>
</tr>
<tr>
<td>PROPERTY</td>
<td>Insert</td>
<td>0.342</td>
<td>1.217</td>
<td>1.097</td>
<td>0.301</td>
</tr>
<tr>
<td></td>
<td>Replace</td>
<td>0.372</td>
<td>4.752</td>
<td>1.205</td>
<td>0.316</td>
</tr>
<tr>
<td></td>
<td>Delete</td>
<td>0.298</td>
<td>1.073</td>
<td>0.939</td>
<td>0.230</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>Insert</td>
<td>0.320</td>
<td>2.430</td>
<td>1.039</td>
<td>0.288</td>
</tr>
<tr>
<td></td>
<td>Replace</td>
<td>0.353</td>
<td>3.853</td>
<td>1.149</td>
<td>0.304</td>
</tr>
<tr>
<td></td>
<td>Delete</td>
<td>0.288</td>
<td>1.067</td>
<td>0.909</td>
<td>0.226</td>
</tr>
<tr>
<td>ENTITY</td>
<td>Insert</td>
<td>0.302</td>
<td>2.102</td>
<td>0.968</td>
<td>0.282</td>
</tr>
<tr>
<td></td>
<td>Replace</td>
<td>0.324</td>
<td>3.237</td>
<td>1.046</td>
<td>0.286</td>
</tr>
<tr>
<td></td>
<td>Delete</td>
<td>0.325</td>
<td>1.156</td>
<td>1.029</td>
<td>0.260</td>
</tr>
<tr>
<td>Dependency</td>
<td>Add</td>
<td>0.005</td>
<td>0.047</td>
<td>0.059</td>
<td>0.021</td>
</tr>
<tr>
<td></td>
<td>Delete</td>
<td>0.019</td>
<td>0.184</td>
<td>0.076</td>
<td>0.039</td>
</tr>
</tbody>
</table>

Table 6.7 Fundamental interpreter operations timings for each test platform.

Figure 6.3 shows how long it takes to add and remove a dependency for any component. The actual time taken to perform this operation is dependent on the number of existing dependencies on the component and its position within the dependency list. The results shown here are, therefore, the best case results.
Figure 6.2 Basic interpreter overheads for three primitive types: a) Element; b) Property; c) Entity.

6.3.5 State Management

The five operations that are performed on a component’s state are those for instance control: construct-destruct, and those needed for state encoding: size-pack-unpack. The duration of these compute-bound operations on an integer, real or boolean is
shown in Figure 6.4 (an empty string increases the time for these actions marginally due to its slightly larger size). There is a linear relationship between state size and operation performance, and the time taken to complete any operation is extended if the component is an array.

---

**Figure 6.3** Cost of adding and removing a dependency on a UML component.

---

**Figure 6.4** Fundamental state operations on an Integer/Real/Boolean and their cost on each platform.

To examine the impact that state size has on the performance of each operation, the length of an array of integers was varied and resultant times recorded. Figure 6.5 State operation costs based upon state size (Pentium).is a graphical representation of the results whilst Table 6.8 details the time increase of operation execution if one element is added to the array.
The performance of the state operations on an empty element compared to that of an element with a single property (an integer) may be viewed in Figure 6.6.
Unsurprisingly the difference is equal to that of a single property (Figure 6.4), therefore the computational cost of managing an element may be calculated by totalling the costs of the individual properties contained therein, added to the basic element overhead.

Similarly, nested elements produce predictable results (Figure 6.7), each level comes at the price of a single element’s overheads.

![Figure 6.7 State operations on elements with one to ten properties (Pentium).](image)

The cost of reinterpreting any part of a UML definition may be estimated by determining the differences between the current and new definition. Those parts that are now obsolete must have their state destructed and then the relevant portion of the data structure removed. New component definitions are added in the normal way whilst those components that are redefined require partial (or complete) state destruction, re-interpretation and re-construction.

When preparing state information for transmission its total size is estimated and a buffer is allocated into which the state is packed. The receiver of the state unpacks the transmitted buffer into its data structure. If the sender or receiver uses big-endian byte ordering, then a byte swapping operation is performed when packing or unpacking respectively. Figure 6.8 shows the performance of each of the test platforms when the three state encoding operations are performed on primitive types of the same size. Although Reality must always byte-swap its state, the performance of these operations without byte-swapping is shown for comparison purposes.
Figure 6.8 Costs of state sizing/packing/unpacking a Boolean/Integer/Real on all the test platforms.

Figure 6.9 State encoding operation overheads for a String of 40 characters.

The time it takes to perform these same operations on a string with 40 characters is about 10 times slower than for the other primitive types (Figure 6.9). The extra time is consumed by the larger amount of data that must be copied into the buffer. There is no real difference between the performance of the byte-swapped operation and the normal version because character strings are not swapped in any way, only the integer that is used to hold the size of the variable length string.
6.3.6 Summary of UML Analysis

The amount of memory used by the interpreter is just as important as how fast it can interpret and execute UML code. The size of the textual UML definition is of interest since it may be sent between processes and thus affects communications performance. Whilst compression techniques can greatly reduce the space used by such descriptions, the computational overhead is prohibitive unless specialised hardware is available to accelerate the compression and subsequent decompression process. A compromise could be the transmission of tokenised code but this would reduce readability.

The cost of interpreting such definitions was presented in the previous sub-sections. Not only may the computational cost of managing the interpreter's internal data structure be estimated, but also the memory it occupies by applying the simple equations and empirical data in sections 6.3.2 and 6.3.3. It has been shown that there is a simple relationship between the time taken to process a component's state, its size and its structure. Such a relationship enables predictions to be made about the time required to manage state information.

6.4 PML

Performance evaluation of the PML can be conveniently broken into two parts: message transmission and message reception. Although the time taken to send a message is somewhat dependent on the processing done at the receiver, they can, for the most part, be treated separately. All of the charts in this section are based upon message size and therefore have only been calculated up to the largest message size currently supported: 20 Kbytes. In most cases, the performance of only one platform will be presented although the full suite of benchmarks were executed on all platforms. The equivalent graphs for the other platforms can be found in Appendix D.

---

3 This is an arbitrary limited imposed in the prototype and does not reflect an operating system limitations.
This section examines the performance of both the QNX IPC and TCP/IP mechanisms as utilised by the PML. Only Gateway supported TCP/IP under QNX, this is unfortunate because it is also the slowest of all the test platforms. However, the relative performance of these two mechanisms can still be compared.

6.4.1 Transmission

Each communication mechanism shares a common need for a separate mailer process used purely for message transmission. In addition to the general cost of each IPC mechanism, the impact of communications to the mailer and the effect of transmission over Ethernet are examined.

\[\text{Figure 6.10 Relative overheads imposed by a pipe on each test platform.}\]

6.4.1.1 Pipes

Figure 6.10 shows the time taken to transfer messages with different sizes along pipes on each platform used in the evaluation. Under QNX a pipe has a buffer size of 5 Kbytes, therefore when a message length exceeds a multiple of this buffer size, an extra `read()` system call is required. This extra operation is reflected in the chart as small jumps in transfer time at 5, 10 and 15 Kbytes. Despite having the faster CPU clock speed, Server has the worst performance. This can be attributed to having a slower internal bus speed than Gateway, whereas Pentium benefits from having a
much faster CPU. IRIX uses a pipe buffer size of 10 Kbytes but the test results are too noisy to identify the relevant shifts in performance.

6.4.1.2 QNX IPC

Figure 6.11 shows a simple breakdown of the tasks performed by the PML in order to send a message using QNX IPC. The administrative overheads include filling the transmission buffer and, for remote communications, establishing and destroying a virtual circuit. The time taken to complete the actual Send() system call is also shown, including the time that the remote PML needs to receive the message and unblock the sender. All message sends must be sent to the mailer via a pipe (section 5.3.5.4); the delay caused by this is also shown and is added to the other overheads to produce a total send time. The proportion of time used by each of these tasks is similar for each platform.

6.4.1.3 Latency

A comparison of the different QNX platforms used to run the prototype and their impact on message transmission latency is shown in Figure 6.12. The plotted data includes the latency introduced by the pipe. Figure 6.13 shows the difference in latency between local and remote inter-process communications. Unsurprisingly, on Pentium, communications with Gateway have the highest latency since it has the slowest processor. At the other extreme, when examining the same properties on Gateway, the longest delay is experienced when communicating with Server (Figure 6.13b). This result is foreseeable since it is the slowest combination of CPUs within the three systems.

There is, therefore, a large difference between the latency experienced when sending a message to a local process and one on a remote node. On Pentium this magnitude ranges from 6-30 times longer for a remote communication, whilst Gateway experiences anything from 4-15 times greater delay.
Figure 6.11 Deconstructed PML overheads for sending a message under QNX (Server).

Figure 6.12 PML message transmission latency between local processes on the QNX platforms.
Figure 6.13 PML message transmission latency between remote processes on:
  a) Pentium and b) Gateway.

6.4.1.4 TCP/IP

Figure 6.14 shows the time taken by each of the main stages to send a message using
TCP/IP (under both QNX and IRIX) to another process on the same node and an
absolute total which includes the pipe overhead. Establishment of a connection to the
destination process is the most expensive stage: approximately 11 ms on Gateway and
1 ms on Reality (shown as dashed lines). The default TCP transmit buffer size under
QNX is 7300 bytes and the default receive buffer size is 8192 bytes, therefore the
TCP buffers were set to accommodate the largest message size under QNX to avoid
unnecessary message segmentation. This action alone accounts for around 4 ms of
the total time required for buffer control. The IRIX buffer sizes default to 64 Kbytes and were not modified for the test.

![Graph a)](image1)

![Graph b)](image2)

**Figure 6.14 PML message transmission times for TCP/IP: a) under QNX (Gateway); b) IRIX (Reality).**

The large performance difference between the QNX implementation of the TCP/IP protocol stack and its own proprietary IPC mechanism is shown in Figure 6.15. The reasons for poor TCP performance are discussed in section 6.4.4.
6.4.2 Reception

The tests used in this section are based upon the same methodology used in the previous section for message transmissions and are decomposed into their constituent tasks. Calculations are simplified, however, since the latency introduced by a pipe is not present when receiving a message.

6.4.2.1 QNX

There are three basic tasks that are performed when receiving a message using QNX IPC: actual message reception into the receive buffer, unblocking the sender and extracting the message from the buffer. To minimise the transmission latency, the sender is unblocked directly after the buffer has been filled. The average time for this sequence of events is shown by a dashed line. Figure 6.16 shows how much of the total receive time is used by the administration overheads. A slight trend towards longer durations is visible in the administration tasks as the message size increases.

In a similar manner to message transmission, receiving a message from a remote process takes a lot longer than from a local process (Figure 6.17): on Pentium approximately 6 times longer.
Figure 6.16 Breakdown of a PML message receive under QNX (Gateway)

Figure 6.17 Comparison between receiving messages from local and remote processes (Pentium).
6.4.2.2 TCP/IP

Figure 6.18 depicts the duration of the major stages required to receive a message using TCP/IP through polling and blocking. When blocking for a message, the `accept()` call is issued immediately; when a connection is made, the message is spooled into the receive buffer and then the message is extracted from it. For a polled receive, the `select()` system call is used to check for pending connections and
accept() is only called when there is a connection waiting. The chart deceptively shows that a blocked receive is faster than a polled receive since it does not include the (potentially very long) period when the process is waiting. Under QNX, the receive buffer was increased in size in order to accommodate the largest possible message. Again we see that Reality outperforms Gateway by approximately 10 times for a blocked receive and 2-3 times for a polled receive. The unusual sharp decrease in performance experienced at about 9 Kbytes with IRIX TCP/IP is consistently repeatable. The only explanation that the author can offer is that this is the result of some internal buffering in the IRIX socket daemon and may be connected to the poor TCP/IP performance experienced (section 6.4.4).

6.4.3 Throughput

A useful metric is the amount of data that can be transmitted in any given period of time – throughput. This section discusses two forms of this metric: \textit{local throughput} which refers to data transfer within a machine and \textit{network throughput} which refers exclusively to data transfer between machines.

6.4.3.1 QNX

Comparison of local throughput is straight forward when all platforms use the same operating system. The maximum throughput at a given point can be calculated as follows:

\[
\text{maximum throughput} = \frac{1000}{(\text{send time} - \text{receive time})} \times \frac{\text{message size}}{1024} 
\]

Using QNX IPC, it is necessary to subtract the time it takes the receiver to unblock the sender from the actual send time (not including administration overheads). Then it is just a matter of converting the result into Mbytes per second. Figure 6.19 shows the maximum amount of data that can be sent within each system based upon message size. If throughput was limited by bus speed we would expect to see Gateway slightly outperforming Server and an increased throughput for Pentium with its PCI bus. As it stands, however, internal throughput seems to be compute bound. The actual
throughput will be less if the sender and/or receiver are not getting as much CPU as they need.

![Graph showing local throughput within each node using QNX IPC.](image1)

**Figure 6.19** Maximum local throughput within each node using QNX IPC.

![Graph showing network throughput between node pairs.](image2)

**Figure 6.20** Maximum QNX network throughput between node pairs.

All of the QNX platforms share the same physical LAN and each has the same make of network card connected to the same type of internal bus. Therefore, it should be possible to estimate maximum network throughput using the same technique used for local throughput. Figure 6.20 represents the estimated maximum network throughput.
between each possible node pair. The calculation was performed each way on the link, e.g. Pentium to Server and Server to Pentium, and the result averaged to simplify this chart. The results reinforce the conclusion that throughput is compute bound: the two fastest machines have the highest throughput, followed by the fastest and slowest and then the two slowest machines.

The uncharacteristic drop in performance when the message size reaches 4 Kbytes is caused by a large number of out-of-window collisions being generated by faulty Ethernet cards. In fact, throughput should rise dramatically as message size increases and start to level out at around 6 Kbytes. This problem had not been noticed until these tests were run.

The largest message transfer gives Pentium a network throughput of 0.762 Mbytes/sec which may also be expressed as 6.096 Mbps. On a 10 Mbps Ethernet network this is a high utilisation rate which may be attributed to QNX’s lightweight protocol and few collisions due to the controlled manner in which the tests where executed. This figure is, in fact, 0.1 Mbytes/sec lower than the manufacturer’s own performance data for a 20 Kbyte message and is almost certainly due to the aforementioned problem.

6.4.3.2 TCP/IP

The local message passing throughput for QNX TCP/IP is on the same scale as that of QNX IPC network throughput. Reality, however, matches the top QNX message passing performance (Figure 6.21). A meaningful value for the network throughput between these two machines could not be obtained because they are located two miles apart and are separated by two Ethernet LANs, a large FDDI MAN and many routers. The traffic on these networks is generated by machines scattered throughout the university.
6.4.4 TCP/IP Performance

The way that TCP is used by PML has highlighted a problem with this protocol. A connection passes through various states during its lifetime, the last of which is TIME-WAIT. The connection spends long enough in this state to ensure that the remote end has received the acknowledgement of the connection termination request.
and that all segment\textsuperscript{4} duplicates have expired (Postel, 1981a). This period is twice the Maximum Segment Lifetime (MSL) which is the time a TCP segment can exist in the internetwork system. MSL has been arbitrarily defined as 2 minutes although, as noted by Jacobson \textit{et al.} (1992), TIME-WAIT has more to do with the round-trip time for the connection than anything else. Regardless, if TIME-WAIT is not long enough it is possible for old duplicates to infect a new connection (Braden, 1992).

Jacobson \textit{et al.} have noted that this state could cause an indirect performance problem if an application repeatedly closes one connection and opens another at a very high frequency. The current limit of available TCP ports on any host is $2^{16}$. PML establishes a connection every time a message is sent, consequently there is a rapid build-up of connections in the TIME-WAIT state. For simulations with many entities this can soon produce thousands of connections in the time-out phase. The results shown in Figure 6.18 were obtained by ensuring that the benchmark for each message size started when there were no connections still timing out. Figure 6.22 shows what happens if connections are made with others still in time-out. When the message size is small the benchmark program has a short execution but creates a lot of connections. Up until around 8 Kbytes this happens at a rate faster than time-outs occur, but afterwards more connections time-out than are established which results in increased performance. Under QNX the time-out period is around 30 seconds whilst IRIX uses a period around twice that which means that this problem is less pronounced with QNX.

The only way of improving performance using TCP is to maintain fixed connections between key processes but at the price of increased memory and computational overheads on the part of the PML (section 5.3.5.3). Before any decision is taken on whether or not to pursue this solution, it would be prudent to investigate the potentially more rewarding problem of a reliable datagram service (section 6.6.3).

\textsuperscript{4} The user message data is broken into segments by the TCP when sent along a connection.
6.4.5 PML Summary

The tasks of sending and receiving messages using two IPC mechanisms have been broken down into their constituent parts and analysed. QNX IPC is very lightweight and subsequently outperforms TCP/IP when running under QNX. The faster processing power available to the IRIX implementation shows that this protocol can be used in systems of this nature. However, its performance is rather unpredictable, especially when there is a high connection turnover rate. It would seem, therefore, that TCP/IP is best used for communications between nodes in a USS and that an alternative local IPC mechanism is used, e.g. based upon shared memory.

When lightweight threads become readily available it will remove the need for the physically separate mailer process and thus the latency introduced by the pipe. This would improve transmission times at the most by between 0.5 ms and 3.5 ms depending on the platform.

The dramatic difference between message passing performance locally and remotely using QNX IPC was shown in Figure 6.13. This is due to a throughput difference of over 10 times and emphasises the importance of reducing to a minimum the amount of data that is sent between machines. In this test case the machines were only located 1 metre from each other, if they had been further apart, e.g. separated by routers, the results would have been even worse.

6.5 Simulation Execution

A UM provides a number of services, most of which serve to progress the simulation as fast as possible. There are a number of factors that dictate performance:

- Number of entities.
- Number of managers.
- Number of monitored components.
- Frequency of state updates from each entity.
- Size of the state updates.
The contributions made by each of these factors is application dependent and can vary quite substantially. For example, an architectural walk-through may have a large number of entities but they will be predominantly static and therefore produce few state updates. On the other hand, a highly dynamic simulation such as birds flocking will require constant state recalculation. To fully explore all of the possible options would take a very long time and it is unclear what benefits such a varied and non-specific analysis would produce, therefore a more pessimistic approach has been taken.

The core sequence of events for one simulation cycle are as follows:

1. Send an update notify message to each entity and manager in the system.

2. Each entity sends its state updates to its local UM.

3. The UM forwards the state messages to interested managers (and other UMs).

4. When all state updates have been sent, each entity sends an update complete message to its local UM.

5. When all entities have completed the manager is informed and performs its processing.

6. The UM waits for all managers (and slave UMs) to finish their work before starting again at stage 1.

The factor that will have the most impact on performance is the amount of state updates that the UM must handle. This is directly related to the number of interested components and managers in the system. Through examination of a worst-case scenario, a more insightful and stable picture is presented of an architecture that attempts to reduce state flow as much as possible.
Figure 6.23 Activity breakdown of a UM when there is one monitored component and: a) no managers; b) 1 manager; c) 2 managers (Pentium).
To test the affects of state updates, each node was stressed using increasing numbers
of entities, each of which modified its state every simulation step and was monitored
by an increasing number of managers. The majority of the charts in this section show
the duration of the simulation step as it is affected by entity numbers. Under QNX,
the number of entities that could be used was limited by the amount of memory
available on each node. The memory consumption varied depending on the amount of
state information each entity had and the number of managers in the simulation that
were monitoring that state. The universe definition consisted of one and two
properties of integer type for the one and two monitor cases respectively. The sizes
of the actual messages used in the tests were very small, averaging < 100 bytes. This
is because the state transmissions for the monitored components only contained data
for one or two integers. If the simulation protocol overheads can be established then
the impact that an increased state size would have on performance can be
extrapolated from the knowledge of its structure (section 6.3) and the increased
message sizes (section 6.4, section 5.3.3). The source code for the benchmark
manager and entity used in the tests can be found in Appendix B.

Several configurations of a USS were examined:

1. Single node (all nodes were tested in this configuration).

2. Two nodes with the Pentium occupying the master node role and
   Server acting as the slave.

3. Three nodes - the same as configuration 2 but adding Gateway as
   another slave.

The test results obtained with these configurations are presented below and are
followed by an examination of the entity migration mechanism.

6.5.1 Single Node

When there are no managers in a system, there is not a need for entities to send state
updates. Consequently only update notification/complete messages are sent to the
RM and update messages transmitted to each entity. The idle time shown in Figure
6.23 represents the time spent waiting for the entities to inform the UM that they have completed their update. When a special manager is introduced and registers interest in one component (1 monitor) a considerable time is spent relaying each entity's state to it. This does not, of course, affect the amount of time spent idle but does reduce it relative to the total simulation step duration. When there are two managers interested in the same component the state must also be sent to that manager (Figure 6.23c) which, in this case, means more time is spent relaying state than sending the update messages. The time needed to send the update complete messages increases slightly with each manager but is so small that it barely registers on these charts and is therefore not shown.

Figure 6.24 shows equivalent charts where there are two monitored components. When the case with a single manager and two monitors is compared with that of a single manager and one monitor, it is clear that the time spent sending state information has doubled. The same is true for the equivalent cases with two managers.

Two different perspectives on these results are shown in Figure 6.25: firstly in terms of simulation steps per second and, secondly, as a workload relative to the case with no managers. An extra case is presented here, that of three managers and 1 monitor whose performance is matched by the 1 manager, 2 monitors case. This may be explained by examining the messages sent in each circumstance.

If there are \( x \) managers, \( y \) monitored components and \( z \) entities, then \( yz \) state updates are sent by entities to the UM and \( xyz \) state messages received by managers in total each step. For 10 entities this results in 10*1 messages originating from entities and 3*10*1 messages sent to managers in the former case - a total of 40 message transmissions. Applying the equations to the latter case, 20 state messages are sent by entities and 20 messages received by the manager. Therefore the same amount of bandwidth (40 state messages) is being used every simulation step resulting in the same simulation rate. The slight performance discrepancy visible in the charts can be attributed to the different numbers of update notification/complete messages that are
sent in each case and the difference in total manager overheads. In this case, state transmissions are the largest performance limiting factor.

Figure 6.24 Activity breakdown of a UM when there are two monitored components and: a) 1 manager; b) 2 managers (Pentium).

The UM's idle time is the sum of the total time spent polling for an incoming message (because there are still pending internal events in the action queue) and the time spent blocked, waiting for a message since there is no other work to do. Figure 6.26 shows this idle period as a percentage of a simulation step (which gets longer as the number of entities increases). For small numbers of entities the amount of time spent idle is high but it soon settles into a consistent rate as the number of entities and system workload increases. When there is no state to forward, the UM is idle for around
30% of the time, but when there is one or more managers in the system, the UM idles approximately 16% of the time.

![Graph a) Steps per second vs Number of Entities](image1)

![Graph b) Percentage of basic performance vs Number of Entities](image2)

Figure 6.25 Effects of various factors on simulation rate: a) steps per second; b) percentage of basic performance (Pentium).

All of the results presented here are from the tests performed on the Pentium platform. The results for the other platforms have the same relative proportions but are on a smaller scale. Figure 6.27 shows how the baseline UM performance (0 managers, 1 monitor) compares with the equivalent configuration on Gateway. Pentium consistently performs on average 3 times faster than Gateway. Server's results (not shown) are very similar to that of Gateway's, reflecting their comparative computational power. Complete charts may be found in Appendix E.
6.5.2 Two Nodes

To examine the performance effects of a multi-node USS, the same tests used in the single node trials were repeated with the entities distributed amongst the nodes. The decision of whether to allocate an entity to one node or the other was based upon a CPU rating derived from the single node results obtained previously. For example, in the case with no managers, the total simulation time for 31 entities on Pentium is ~38
ms, whereas this same time is used by 9 entities on Server. This would give a CPU rating for Pentium of just over 3 times that of Server's, a figure backed up by the CPU performance figures given in Table 6.1.

![Graph a) SUM Simulation](image)

![Graph b) MUM Simulation](image)

**Figure 6.28 Activity breakdown of UMs in a master-slave configuration with no managers: a) SUM (Server); b) MUM (Pentium).**

In this and subsequent multi-node tests the fastest node was used to run the MUM, the activity breakdown of which may be found in Figure 6.28b. The total simulation time for each node is identical since the SUM must wait for the MUM to send it an update notification message before it begins each simulation step. The stepping effect is caused by the changes in entity distribution which is measured by the scale on the right hand side - at most 40 entities were used system-wide. The somewhat irregular shape and downward tilt of the steps is a reflection of the error in the distribution
algorithm. That is, whereas an optimum distribution may require fractional parts of an entity to be distributed in order to keep the workload exactly balanced, only whole entities can be moved.

![Graph](image-url)

Figure 6.29 Activity breakdown of UMs in a master-slave configuration with 1 manager on the master node: a) SUM (Server); b) MUM (Pentium).

A large portion of the SUM's time is actually spent waiting for the MUM to start the next simulation step - 14 ms in this case. The SUM notifies the MUM that it has completed its processing for that step and, when all the MUM's local processes have finished as well, the MUM sends the next update complete message. The waiting time is therefore the sum of two message transmission latencies and some processing with the exception of one condition. It is true that a significant portion of the idle
time can be attributed to the waiting period. However, it is possible for the SUM to wait for a period greater than its idle time if the SUM should be starved of CPU - a situation that may occur in a heavily loaded simulation. This aside, if the waiting period is subtracted from the simulation time series, the product is the equivalent of a single node simulation.

The same test conditions were used to introduce a manager on the node with the MUM and the largest number of entities (Figure 6.29). This reduces the state updates sent over the network to a minimum, i.e. the few entities on the SUM send them to the MUM. Initially, when there are no entities on the slave node, performance is identical to the previous case. However, as soon as an entity is allocated to the slave the latency of a state update is incurred. This, added to the additional message processing on both nodes significantly increases the duration of the simulation step.

The effect of placing the manager on the slave node rather than the master node may be seen in Figure 6.30. When there are 40 entities in the system, 31 state updates must be sent across the network to the SUM and then forwarded to the manager. The master's chart (Figure 6.30b) shows that the MUM spends most of its time idle, waiting for the slave to process all the state information. The chart is somewhat deceptive, however, since the state time does not include the message transmission latency which would put it close to the total simulation time and reduce the idle time appropriately. Introduction of a second manager on the master node increases total simulation time by about 100 ms (when using 40 entities) since state information is now also sent from slave to master.

It is clear, therefore, that not only is computational power an important consideration when distributing entities\(^5\), but also the inter-node communication overheads and the location of special managers. With the technology used in this prototype, the network is by far the most limiting factor.

---

\(^5\) Without resource dependencies, such as input devices.
6.5.3 Three Nodes

Figure 6.31 shows the task durations on three nodes as the simulation is distributed amongst them. In contrast to the equivalent case with two nodes (Figure 6.28), the wait time has risen to 20 ms and the overall simulation time by 10 ms. The MUM's largest workload has been lightened by 8 entities which have been spread between the two SUMs. The added processing time incurred by the extra node has caused the MUM's increase in simulation and wait times.
When a manager is added on the master node, the total simulation step time degrades to a maximum of 218 ms which is \(\sim 40\) ms more than the equivalent case with 2 nodes (Figure 6.29). However, as Figure 6.32 shows, when the manager is allocated to a slave node, the overall system performance is identical to the master-slave case depicted in Figure 6.30. On an individual basis, the first SUM (on Server) is managing one less entity than previously which results in slightly lower state management times.

Unlike the other slave node, the SUM’s waiting period is greater than its idle time, indicating that there are other processes on that node that have more urgent need of the CPU. The idle time is smaller because some of the time that the SUM would have spent idling was consumed when it was waiting for its next timeslice. Therefore, despite communication latency hindering performance, this three node configuration, with a manager on Server, is as efficient as the master-slave case presented in the previous section.
Figure 6.31 Activity breakdown of UMs in a master and 2 slaves configuration with no manager: a) SUM1 (Server); b) SUM2 (Gateway); c) MUM (Pentium).
Figure 6.32 Activity breakdown of UMs in a master and 2 slaves configuration with a manager on slave node Server: a) SUM1 (Server); b) SUM2 (Gateway); c) MUM (Pentium).
6.5.4 Entity Migration

In order to demonstrate entity migration it is necessary to have some way of estimating resource usage for each entity. A measure is not meaningful unless it is measured with reference to a fixed time span, i.e. a simulation step. Since full scheduling and RM functionality had not been implemented, only CPU usage was monitored and a suitable step duration threshold specified (sections 5.6.5, 5.7.3). Every step the RM obtained the current CPU usage for each process on the node and if the total consumption for all entities exceeded the threshold, the most expensive entity was volunteered for migration. The processor usage for the RM and the UM was not included in the total to simplify the charts. Unlike the entities used in the previous tests which had a uniform workload, a random element was programmed into each entity which would trigger a gradual increase in CPU usage. After peaking, this consumption would diminish until the entity’s original workload level had been reached. In all cases a total of 40 entities system-wide was used and the threshold was set to 65 ms (with the intent that 70 ms would not be reached), beyond which a migration is required. The MUM does not actively load balance in these tests which rely on the passive mechanism triggered by a threshold violation.

Figure 6.33 shows two time series which represent the workloads of the two nodes in the test system and the number of entities present on each node (measured using the scale on the right). The peaks on Server are much higher since it has the slower CPU and at the 30th step exceeds the threshold resulting in a migration. The only place for the entity to go is the master node (Pentium) which already has an entity on the downward slope of a brief workload increase. Both nodes progress as other entities experience increases in workload. The double peaked feature at step 175 represents the product of the workload of two entities, one decreasing, the other increasing. After 250 steps, the nodes have gone from the same starting workload to one that differs by 6 ms. This does leave room for moving a few entities around to improve the load balance if the MUM was actively load balancing.
Figure 6.33  Single entity migration within a two node system (Pentium/Server).

Figure 6.34  Multiple entity migration within a two node system (Pentium/Server).

Another example of entity migration on two nodes is shown in Figure 6.34 where two entities on Server cause the threshold to be exceeded. This time an entity also pushes Pentium over the edge at step 120 and it is migrated to the slave node.
Figure 6.35 Multiple entity migration within a three node configuration: a) migration between slaves only; b) migration between all nodes.
The charts in Figure 6.35 show entity migrations occurring in a three node system. In Figure 6.35a two migrations are required from Gateway and in both cases the chosen target node is Server. Each time the target node's CPU also exceeds the threshold, this is due to the fact that the state construction performed for each entity directly after migration is more expensive than the update function. For this reason, after a process has been created a four step period\(^6\) is used to wait for the CPU consumption to settle down to normal levels. If this hysteresis period was not in force then the target node would immediately reject the new entity; an action which could be repeated any number of times resulting in the entity bouncing between nodes and thus destroying system performance. The second migration shown in Figure 6.35b is from Pentium to Gateway which is clearly a mistake on the part of the load-balancing algorithm. Even after the resting period, CPU consumption is far too high and the entity is migrated to Server.

The workload patterns in these tests were contrived but clearly demonstrate the migration mechanism. It is also clear that more comprehensive information must be used to determine the target node in order to avoid misallocations, i.e. a full RM implementation is needed (section 5.7). If some allowance was made for a short burst of CPU when the new entity is constructing, it would be possible to remove the current four step settling period.

### 6.5.5 Process Activity

Currently the time between starting a process and it reaching alive status is more in the region of hundreds of milliseconds rather than a few milliseconds. The most intensive part of this time is the creation of the main component process and the mailer. Following this the allocation of the UPID must take place and then the process' internal initialisation which can vary depending on its purpose, i.e. manager or entity. The actual creation time for a process also depends on the number of other processes starting at the same time. For example, when initiating a simulation with 40

---

\(^6\) Discovered by empirical means.
entities, all entities and managers may not reach an active state until 30 seconds after
the UM was started. Creation of a process on a slave node by the MUM is further
confounded by the communications latency.

From the UM's perspective, the termination of an entity is quicker because the actual
process termination is faster and the administration overheads are comparable with
creation, e.g. informing managers of an entity's death.

6.5.5.1 Benchmark Entity

An integral part of the entity process creation/termination is the execution of that
entity's specialised construct and destruct functions respectively. The duration of
these functions in the entity that was used in all but the migration benchmarks is
shown in Figure 6.36. Execution of the entity's destruction routine takes longer than
construction because constructing a UML component generally takes less time than
destroying it (section 6.3.5). This does not hold true for Reality in this case,
probably because the unoptimised code for construction is actually slower than the
operations needed to free memory.

Approximately 38% of the time that an entity uses when updating is spent sending the
state data to the UM. The remainder of the time is used by the invocation of the
entity's update function prior to the state transmission which, in this case, merely
toggles the flag to indicate that the component has been modified (although its value
is not actually changed).

Measuring the performance of an entity with the current prototype is somewhat
problematic since the duration of any given task is totally dependent upon scheduling.
As the number of entities grows the variances in measured duration become more
profound; despite getting the same amount of CPU each time. The best way to
measure an entity's performance, therefore, is to restrict the simulation to one entity
such that it is unlikely to be interrupted during measurement. Figure 6.37 shows that
the entity used in the benchmarking is idle for ~79% of the simulation step under
QNX, the update taking at the most a few milliseconds. Again, due to poor TCP/IP
performance, Reality is already idling at 99% along with all the other system components. If the number of entities in the simulation was increased we can expect to see an increase in both the time it takes to send a state update and the idle time of the process.

**Figure 6.36** Construct/Update/Destruct times for the entity used during benchmarking.

**Figure 6.37** Average task breakdown for a single entity.

### 6.5.5.2 Benchmark Manager

In the same way that most of the entity's life is spent idle, so is that of the manager used in the tests when there is only one entity in the simulation (Figure 6.38). The ratio of processing state updates to the total step duration will always be small, because a manager only performs its work at the end of the step when all entities have
sent their updates. With just one entity the manager is idle around 88% of the time under QNX, but an increase in the number of entities will increase the time spent processing updates, the step duration and the idle time.

![Activity breakdown for the benchmark manager.](image)

**Figure 6.38 Activity breakdown for the benchmark manager.**

### 6.5.5.3 Resource Manager

The most intensive activity that the current RM performs is obtaining the CPU usage of each process on its node. The basic overheads specified in Table 6.9 represent the cost of monitoring itself and the UM; the costs of checking CPU usage for an entity or manager is also given. Usage of the PML mailer process is included in the calculated resource ratings. It is clear that this process is quite computationally expensive if performed every simulation step - as it was in the migration benchmarks. However, the current version does not attempt to perform any usage predictions that may be used to reduce the frequency with which this monitoring is required.

<table>
<thead>
<tr>
<th>Overheads</th>
<th>Pentium</th>
<th>Server</th>
<th>Gateway</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic (ms)</td>
<td>5.72</td>
<td>20.85</td>
<td>19.17</td>
</tr>
<tr>
<td>per Process (ms)</td>
<td>0.8</td>
<td>2.705</td>
<td>3.84</td>
</tr>
</tbody>
</table>

Table 6.9 Time penalties incurred when RM monitors CPU usage.
6.5.6 Simulation Execution Summary

This section has concentrated on the performance of the USS as a whole. Each of the test platforms, with the exception of Reality, were examined as single node systems. This provided a basis for evaluating performance when they were combined in two and three node systems. Although the same tests were performed on Reality and Gateway using TCP/IP, due to the problems with its use, each process spent 99% of their time waiting for messages. This issue is dealt with in section 6.6 which looks at improving the prototype's performance.

It was found that the most limiting factor in a distributed configuration was the network latency and that it had a substantial impact on performance. By carefully allocating processes to nodes, a three node configuration was shown to produce the same performance as an equivalent two node configuration. However, for the same number of entities this was still many times slower than simulating all the entities locally. This is not a common situation since the entities in question did no real work and used little memory. Given computationally more expensive or physically larger entities, distribution becomes a necessity rather than a luxury.

Often entities will consume different amounts of resources at different times which has the net result of producing a variable node workload. By migrating entities from one node to another, the available resources may be utilised to the maximum. To test the migration mechanism, multi-node systems were stressed with a number of entities, each with a variable computational workload. Finally, it was shown that it is possible to monitor resource usage and move entities in order to keep the system workload relatively evenly balanced.

6.6 Improving Performance

Based upon the knowledge gained from the analysis of the prototype presented in this chapter, it is clear that there are a number of improvements that can be made.
6.6.1 Message Elimination

One of the most limiting aspects of the current implementation is the use of point-to-point communications between processes, especially the UM and its entities. Figure 6.39 indicates the percentage performance increase that would be experienced if the update notification messages could be sent to all entities simultaneously rather than sequentially, i.e. an inter-process multicast. This was calculated by replacing the usual linear increase of time for the update task with the time taken to update a single entity; all other overheads were left untouched. The figure shows that a multicast method would produce greater performance benefits as more entities are added to the system. If the chart was extended by testing the method with more entities the performance increase would remain about 150%. The overall effect is not as dramatic with those configurations that transmit more state information. Also, the impact is diminished because each entity must still inform the UM that it has completed updating every simulation step.

However, use of both the update notification and complete messages within a node is actually mimicking the behaviour of a deadline scheduler. When the scheduler triggers the entity to start processing, this may be taken as a cue to begin updating. After sending any state updates that were necessary, the entity would reach its deadline and this would indicate to the UM that the entity had finished updating. Therefore the actions that are currently performed explicitly with messages would be replicated implicitly by the nature of the scheduler. The greater the number of entities, the more time saved each step by eliminating the update complete message (Figure 6.40).
6.6.2 Shared-Memory IPC

QNX IPC essentially copies a block of memory (the message) from one process’ address space into another, therefore implementation of a shared memory based IPC protocol is unlikely to show much improvement. This is not the case when compared against the burden of using TCP/IP for local communications. The same simulation combinations executed on Gateway when using QNX IPC are a lot faster than when
using TCP/IP on the same machine, e.g. 24 times for 1 manager/1 monitor, and 34 times for 2 managers/1 monitor. The idle rates of processes using TCP/IP on both Reality and Gateway are very similar (~99%) and a shared-memory IPC system will likely show similar performance to that of QNX IPC. Given this, it is not unreasonable to use this speed-up factor as a rough indicator of the performance increase we could expect to see on Reality if shared-memory IPC was adopted. Figure 6.41 presents a comparison between the predicted performance and that of the fastest QNX node using the native IPC. Saying any more than that the two machines now present comparable performance would be unwise given the uncertainty of the estimation procedure used.

![Graph](image)

**Figure 6.41** Comparison of estimated Reality performance with shared memory IPC and Pentium using QNX IPC.

### 6.6.3 Multicast

Of course, TCP/IP is still the only available reliable method for communicating between heterogeneous machines on a network. This is also an area that could be optimised through the adoption of a reliable multicast protocol. A MUM could multicast update notification messages to its slaves and its use would also open up the possibility of state multicasts. The Single Connection Emulation sublayer presented by Talpade and Ammar (1995) is designed to sit between an existing reliable transport
protocol and the network layer providing the unreliable multicast capabilities. The presented implementation used TCP as the transport protocol and IP as the network layer. The existing TCP API is utilised as usual but is supplemented by a direct interface to the SCE layer in order to control the multicast-specific variables of the multicast connections. One advantage of this approach is that it is possible to modify the semantics of multicast connections by changing the SCE without affecting the transport protocol. Unfortunately, because TCP is used, this solution also requires that prior to transmission a connection is established from the source of the multicast to the set of destination nodes. After transmission has concluded the connection must be closed. Consequently, in order to make use of this solution, the modifications to PML operation discussed in section 5.3.5.3 would have to be made. Nevertheless, of the solutions to providing a reliable multicast service that the author has seen, this seems like the most promising. In addition, should a more suitable reliable transport protocol come to light, SCE could be adapted for use with it.

The biggest savings that can be made are with the transmission of state information which is the most common and often the largest type of message that is sent. Each inter-process and inter-node communication pathway has a unique monitor ID associated with every component whose state is transmitted along it. This method works well for point-to-point links where the monitor ID is modified as the state is forwarded to all interested parties, but precludes the use of multicast in any form. A possible solution to this problem would be to replace the monitor ID by the component’s absolute name within the UML definition. The implications of this change would be an increase in message size (the absolute name could be potentially very long), and an increase in the amount of time needed to identify the component in each process’ internal data structures prior to unpacking.

The adoption of multicast communications between machines would remove some of the burden from the master USS; it would also use less bandwidth. Consider a system

7 Multicast between processes on a node may be simulated through a shared-memory buffer that is monitored by all processes.
with a master and two slave nodes: currently state information from a slave is sent to
the master which forwards it to the other slave (if needed). With a multicast only one
message would be required which would reach both nodes simultaneously. This cuts
the required bandwidth by half and as message sizes and the number of slaves
increases, so do the savings. In addition, it is possible that the component dependency
list could be used to form a multicast group for those machines interested in its state
updates. Although this would not reduce bandwidth consumption, it would ensure
that any node not interested in the multicast did not waste time processing the
message. Using a shared-memory emulation of multicasting, such gains as these
could also be experienced by processes on the same node.

The application of this technique promises to yield significant performance increases
but the computational cost of supporting it is uncertain. It is, however, an area
worthy of further investigation.

6.6.4 Accounting for Latency

Section 6.5.2 presented a situation where the slave node spent a considerable amount
of time waiting for the next simulation step to begin. Although it is not possible to
eliminate the time the SUM spends waiting, its counterpart in the MUM may be
removed if the slave's workload is reduced such that it finishes its work earlier.
Figure 6.42 shows the current situation on the left hand side where the MUM (with an
identical workload) has to wait for the SUM to respond before it begins the next
simulation step. If the SUM's workload is reduced by 14 ms from 30 to 16 ms then
the MUM no longer has to wait, thus increasing the simulation rate. In the case
presented in Figure 6.28 this technique would effectively require the reduction of the
SUM's (and the system's) workload by 5 entities. Thus, when the MUM is managing
31 entities, the SUM would be coordinating 4.
Figure 6.42 Accounting for message latency reduces simulation cycle duration.

6.6.5 Increased Bandwidth

Increasing bandwidth would not obviate the need for the technique presented in the previous section, but it is the simplest way of improving performance. The results presented in this chapter were based upon a dedicated Ethernet link with a theoretical speed limit of 10 Mbps. Recently, networking mediums such as FDDI and Fast Ethernet, capable of operating at 100 Mbps, have become widely available. Figure 6.43 shows the impact that using a 100 Mbps link between machines would have on messages sent between Pentium and Server. This prediction is based upon three assumptions: firstly, that the network throughput would experience a seven-fold improvement; secondly, the bus can cope with the increase in required data transfer rate; thirdly, that the same level of utilisation QNX currently achieves would be increased by the same degree (section 6.4.2.2). Whereas it took between 2 and 25 ms to send a message between nodes, this now occurs in 0.3 to 3.7 ms. The result is that

---

8 This assumption is based upon manufacturer's data that states a data transfer rate of 7.4 Mbytes/sec for 100BaseT Fast Ethernet as opposed to 0.9 Mbytes/sec for 10BaseT.

9 An ISA bus cannot match the demands of a 100 Mbps network, whereas a PCI bus will.
latency is now lower than the PML overheads which now account for approximately 55-65% of the total send time.

The impact this would have on the master-slave benchmarks presented is difficult to estimate due to the unknown scheduling factor, but all messages sent were less than 1K in size. This would mean that a few milliseconds would be saved on each transmission. Considering the case presented in Figure 6.29 this would probably result in a latency reduction of around 23 ms (1 update complete + 1 update notify + 9 state updates). Currently all processes idle enough to cope with this decrease in transmission time but consideration of a more complex case would require further investigation.

Figure 6.43 Pentium to Server transmission times when using the current Ethernet link and a predicted Fast Ethernet link.

6.7 Summary

This chapter has presented an evaluation of a prototype USS concentrating on the modeling language, the characteristics of the message passing systems and general simulation performance. A number of points were made in the section summaries throughout this chapter but there are a few observations and aspects worth emphasising.
6.7.1 Living with TCP/IP

Although TCP provides a reliable connection, it uses an unreliable medium (IP). Positive acknowledgement is used to ensure that packets arrive at their destination - failure to do so results in retransmission. The greater the distance between the connection’s two endpoints (and the more routers, etc.) the longer it will take to determine whether a packet has been successively received. The use of a hierarchy to connect nodes (and processes) in a USS and a network of USSs provides a more robust communications mechanism than requiring a single process to communicate with a server over some large distance. Should a link fail then this can be detected far quicker because the distance between nodes is far less. Resolution of this problem can be handled by the node that detected the problem or the sender can be informed and action taken accordingly.

This information also supplements the determination of whether the destination node is still alive. Although routers report when they cannot deliver any given message using the Internet Control Message Protocol (ICMP - Postel, 1981b), they may not be able to detect all such errors. The ability to detect errors is dependent upon the hardware protocol. For example, Ethernet does not acknowledge transmission of packets meaning that a node can be disconnected without affecting the rest of the network. Unfortunately, this also means that with Ethernet it is not possible to detect power failure, etc.

All of the message size tests are dependent upon the Maximum Transfer Unit (MTU) which may be different for all network media. For example, Ethernet has an MTU of ~1500 octets\(^{10}\), whilst FDDI has an MTU of 4770 octets and ATM uses 9180 octets\(^{11}\) (Laubach, 1994). If a message is transmitted greater than the MTU in size then it is fragmented. This fragmentation and corresponding reassembly at the destination

---

\(^{10}\) Some implementations vary from the Ethernet specification.

\(^{11}\) ATM could handle 64K octets but has been limited to 9180 so that it is compatible with the older Switched Megabit Data Service (SMDS) technology (Piscitello and Lawrence, 1991).

290
inevitably incurs some overhead. In addition, the more fragments the greater the chance that one will be lost and the message will be discarded.

TCP/IP has its faults but it is the backbone protocol of the Internet and, in one form or another, it is here to stay. It would seem prudent, therefore, to find some way of working with it.

The Hyper Text Transfer Protocol (HTTP) is used by the World Wide Web (WWW) to retrieve distributed objects. HTTP uses TCP as the transport layer enabling WWW servers and clients to communicate. Every time a request is sent from client to server a connection is established, e.g. loading a new page, new icon/in-line image, etc. As the author discovered when evaluating the performance of the PML, this is an inefficient way of using TCP which is designed to handle data transfer over connections established for long periods of time, e.g. file transfer. Spero (1996) presents a detailed summary of the problems with the way in which TCP is used by HTTP, including TCP’s TIME-WAIT state (section 6.4.4). A proposed solution to these problems is the incorporation of a simple multiplexing protocol to be used with HTTP, enabling multiple requests to be dealt with on a single connection (Gettys, 1996).

The advantages of shared memory IPC over TCP as a local IPC mechanism have already been discussed. However, permanent connections could be established between key components on each node using TCP/IP, for example MUM to all SUMs and between systems, which are the links that need to be optimised the most. The price for this change is increased complexity within the PML which now has to handle two different types of connection. Nevertheless, applying TCP/IP in this manner would be more efficient than the way it is used now and is the equivalent solution to the multiplexing protocol mentioned earlier.

6.7.2 Resource Management

The amount of memory used by each program’s execution image alone is larger than necessary because shared libraries could not be used. Currently each entity process
under QNX requires approximately 256 Kbytes of memory (section 6.2.3); with shared libraries this could be reduced to the order of 6 Kbytes. Around 330 Kbytes of USS shared libraries would be shared amongst all processes in addition to ~55 Kbytes of system libraries.

The memory needed to store the UML definition and its instance data is quite large. Unfortunately this is the price that must be paid in order to effect modifications at run-time with as little disruption as possible. There is a relationship between definition structure and the amount of resources that any given operation consumes. This type of information could be used by the interpreter to predict how long an operation will take or how much memory a definition will require. Accurately predicting resource consumption aids the RM in its work.

The total simulation workload is unlikely to remain the same throughout the simulation and local fluctuations are to be expected. The migration mechanism presented is currently in a primitive state but adequately demonstrates the benefits such a technique can have on system loading. More resource utilisation information is needed so that better decisions can be made about a node’s loading and predictions of an entity’s workload.

6.7.3 Scaleability

A detailed analysis of the PML proved that communicating between machines is many times slower than between processes on the same node. An unexpected problem with TCP/IP was encountered which, combined with its use by the PML, made using it as a local IPC mechanism impractical. Examination of the PML provided a basis for evaluating system performance as a whole and also enabled predictions to be made by modifying key variables such as bandwidth.

A number of simulations were run on single node systems, each one using a different number of managers and an increasing number of entities. The results showed that, in general, more time is spent processing state information that any other type of data. Also, the UM spends a considerable amount of time idle waiting for other processes
to complete their work. This analysis of single node systems provided a basis for evaluating multi-node configurations.

6.7.3.1 Standalone USS Performance

Performance in a USS is dictated by a number of factors (in no particular order):

1. Number of special managers.
2. Location of the special managers/entity processes.
3. Number of entity processes possessing monitored state.
4. Size of the state information monitored by those managers.
5. Transmission frequency of the monitored state.
6. Number of USNs in the USS.
7. Latency/bandwidth of the connections between the nodes.

The more managers that monitor any given part of the whole VE’s state, the more state information that must be sent between processes, i.e. the more bandwidth consumed. The best case is if the manager in question is interested in just the state held by entities on its local node: where the available bandwidth is highest and the latency lowest. The more common case is when a manager is interested in state held by entities that are spread on many nodes within the system. In this case the size of the state information that must be sent to the manager(s) becomes even more important - the more state information or the smaller the link’s bandwidth, the lower the performance. If the manager is on a slave node then an entity on another slave node will send its state update to its local UM, which forwards it to the MUM, then onto the manager’s local UM and finally to the manager itself. If the manager is on the master node then this procedure takes one stage less. If the entity is on the master node and the manager is on a slave node then the procedure is also one step quicker.

The amount of state data sent is dependent upon the frequency of changes to that state made by each entity. It is not possible to calculate in advance what this frequency will be since it is semantic specific. A well designed manager will monitor information that changes on a periodic basis and make use of constraint functions.
These can be used to filter, at source, the state data before it is sent to the manager consuming valuable bandwidth and processing time.

Although the number of nodes in a system and the speed of their communications links plays an important part in performance from a state management perspective, they are also relevant when considering synchronisation. At the beginning of each simulation step the master node synchronises all the slave nodes through an exchange of messages. Using unicast, there is a linear relationship between the time taken to perform this procedure and the number of nodes in the system. Again, this could be partially rectified by replacing the initial master-to-slave synchronisation control unicast with a multicast.

As the reader can see from this list of confounding factors, it is difficult to build a clear picture as to exactly how performance will scale as more managers and/or entities are added to the system. What is clear from the results presented in this chapter, is that performance will fall sharply initially, and then gradually asymptote as more processes are added. However, this could be dramatically scaled down if multicast was used to send state updates to interested managers (Figure 6.39). Not only would bandwidth be saved but the burden on the MUM as a router would be reduced significantly, thus removing what would become a major bottleneck in the system as state information flow increases.

### 6.7.3.2 Networked USS Performance

Performance between USSs is also dictated in a number of ways:

1. Total number of users across all systems.
2. Type of information sent between systems.
3. Method used by the user's shadow process to approximate behaviour.
4. Number of networked systems executing same simulation.

The amount of traffic on the inter-system links is mainly due to two related factors. Firstly, the more users participating in a given simulation, the more user-specific data that must be sent to all systems executing that simulation. Secondly, the amount of
information is dependent upon the type of data being transmitted. For example, low-order information such as position and velocity will be sent almost continuously whereas high-order information indicating changes in behaviour will be sent less frequently. It would seem therefore that the latter would guarantee better performance. Unfortunately this requires a more complex shadow process to interpret the information and do something sensible with it. Balancing the amount of data and the amount of computation a shadow requires to process it is the key to good performance.

With the current hierarchical structuring of systems performance, there will be a non-linear degradation in performance as the number of systems increases. Not only will the effort expended by the MUM in each system be increased due to routing but the time taken to ensure every system gets the transmitted message will also grow. Again, multicast will relieve this problem, allowing a single transmission to reach each system running the simulation. Using this technique, performance should be mainly affected by the number of users in the system, not the transport mechanism.

6.7.4 Distribution at a Price

When the author started this work, distribution seemed like the answer to all the problems regarding limited resources and multi-user interaction. It is now clear that there is a distinct price to pay for distribution and it should only be considered if the advantages outweigh the disadvantages.

Communications latency is presently the largest factor responsible for inhibiting progress of a distributed simulation. In simple simulations there is little to be gained by spreading the load throughout a network of machines because more time will be spent communicating than actually performing simulation work. Only when the simulations become more expensive is this cost offset enough to prove beneficial. The advantages of distribution include the possibility of multiple users. It should be noted, however, that the desire to include more users in a VE may well degrade performance due to the problem just mentioned. At the other end of the scale, if the
presence of many users generates too much state information flow, then it will not matter how much computation there is to perform.

6.7.5 Conclusions

The prototype is not perfect and several enhancements that would improve performance have already been discussed. Some, such as the deadline scheduler, require more specialised operating systems whilst others, such as multicast, need a combination of hardware and software protocols that is not currently readily available. Fortunately, a shared memory IPC mechanism could be implemented now, as could the technique used to account for transmission latency.

The balance between CPU performance, bus speed, memory capacity and network bandwidth (amongst others) is an important one; a well configured system will take all of these into account. For example, if only network bandwidth is increased then eventually there will come a time when the bus may become the bottleneck, or the CPU is incapable of processing data fast enough for transmission. The relationship between these factors is influenced by the software system. Unless analysis of the type presented in this chapter is performed, i.e. at the component and system levels, systems engineers will not be able to deliver the technology capable of supporting distributed VE systems.
Chapter 7

Conclusion

"In my end is my beginning."

Mary Stuart, Queen of Scotland

The final chapter of this thesis begins with a brief reminder of the work presented in the preceding chapters. Following this, the USS architecture is classified using the taxonomy presented earlier on and its most important features are highlighted. A few specific research areas that are relevant to distributed VE systems are also described, indicating the benefits they may provide. Finally, the current trendy topics in the area of distributed VE systems are related to the work presented here.

7.1 Thesis Review

The introduction to this thesis gave a brief introduction to the area of VR, highlighted the emphasis on interactivity, and described the two cornerstones of a system that would support this: real-time and consistency. The services of a real-time system enable the generation of real-time displays which are justified in chapter 3. Consistency reflects the need to ensure that everything in the VE appears in the right place at the right time, to one or more users simultaneously.

Chapter 2 began with an examination of the issues involved in the design of a system capable of distributing VEs. The solutions used by existing systems that have attempted to tackle this complex area vary quite substantially. In order to provide a way of comparing such systems a classification scheme was derived which strove to
categorise each system on the basis of: real-time support, communications, data management, computation management, VE modeling, time management, fault tolerance and security. There is an intricate web of inter-dependencies connecting many of these categories which often makes examination of one difficult without referring to another, e.g. data and computation management working together to provide consistency. However, the author believes that this taxonomy is a good starting point and was applied to the seven distinct systems that were reviewed. The results of applying the classification scheme proposed in this thesis to the USS are presented later in section 7.2.

Chapter 3 questions the current way that VEs are modeled and highlights a particular aspect of human-computer interaction that is not addressed in most systems. To better understand how to model a VE, the structure of the natural environment was examined and several taxonomies of varying levels of detail were presented. Based on these attempts to classify natural and virtual environments, the author presented a suitable definition and abstract model for a VE. Essentially, current modeling practices take one perspective on the thing being modeled and concentrate on one medium, usually visuals. With this approach the model will function adequately until such time as another medium is considered, e.g. sound, or a different perspective has to be taken, e.g. infra-red instead of natural light. At this point the model will falter because some (or all) of the information that is now needed to simulate this perspective/medium will be missing. If a more ecological approach had been taken to modeling, then sufficient information would have been modeled initially such that similar changes would not require extra work. There are obviously practical limits to the amount of information that can be modeled at one time and these are discussed with relation to the modeling process as a whole. When looking at the design process it was noted that an integrated modeling and simulation system would enable development, experimentation and evolution. The ability to develop a simulation on-line provides much greater flexibility than is available with current systems and also a reduced development time cycle. These features will hopefully also encourage the VE designer to explore the different forms the model can take. Finally, evolution referred
to the ability of each entity in the simulation to make changes to the model and create other entities.

Related to the issue of modeling a VE is its display. The purpose of a display is to take raw information from the environment, process it, interpret its meaning, and then present it in a form that enables the viewer to extract some meaning. A good display will permit the natural processing of the presented information and allow the participant to concentrate on the task at hand. A bad display will require the participant to expend extra effort and will probably degrade their performance. The second part of chapter 3 describes how variable-rate visual displays cause problems when judging time to contact with a virtual object. The example given is catching a virtual ball, but it could equally be braking in a virtual car to avoid a collision on a virtual motorway, or attempting to perform in-flight refuelling in a flight simulator. Essentially any task that requires the user to make judgements based on velocity and acceleration/deceleration can be affected if a constant-rate display is not used. Two methods of achieving such a display were presented: one requires special OS support, the other will work on normal operating systems.

Chapter 4 starts with the presentation of the requirements for a USS, a set of realistic design restrictions, and a little more detail on key aspects, e.g. distributed real-time systems. Having settled on a modeling process using specialisation through inheritance in chapter 2, a suitable representation of the VE abstract model is presented. Since the abstract model is derived from our universe, an appropriate naming scheme was adopted based around "universal". A number of existing languages were examined before it was decided that none of them satisfied (or could be modified to satisfy) the requirements of a VE modeling language. The proposed language, UML, can be broken into two halves: data definition and instruction code. The structure of UML is important since it is an integral part of the USS architecture. Although UML code can be passed between USS processes, it could have any syntax or grammar. The data definition, however, influenced the mechanisms used to manage state within the architecture and vice versa. The design is dissected in section 7.2.
After outlining the USS design, a prototype implementation was described in chapter 5. Key to the system is a real-time distributed deadline scheduler which is difficult to implement with current hardware/software technology. The author had, prior to USS development, implemented a far less complex worst-case scheduler at the application level to help enforce a constant-rate graphical display. It was the author’s experience that, even with a special-purpose operating system, use of such a scheduler was problematic due to the difficulty in accommodating actions beyond the application’s control, e.g. network and disk access. Therefore the architecture’s key elements were implemented without the scheduling functionality. The PML is used to provide a common interface to the various OS services that the USS processes require - mainly message passing. Following details of the PML, the structure of the UML interpreter was described, including a detailed explanation of the complex data structure used to hold the model description and its instance data. The remainder of the chapter dealt with each major software component in turn, starting with the UM, and highlighted key aspects of their implementation.

The implementation was evaluated in chapter 6 which started with a characterisation of the platforms used for testing. This was followed by a detailed examination of the UML interpreter, its performance and memory requirements. The impact inter-process communications have on performance was analysed in the section dealing with the PML. The rest of the chapter examined the simulation performance of the system as a whole, in single node, two node and three node configurations. In addition the process migration mechanism was demonstrated using the two and three node configurations. A number of enhancements that could be made to the design and implementation in order to improve the prototype’s performance were also described. The chapter concluded with a discussion of the factors affecting the performance of the prototype and a number of general observations.

### 7.2 USS Classification

Table 7.1 replicates part of Table 2.3 in order to provide some basis for comparison of USS’s features. USS is the only distributed VE system architecture out of those
reviewed that has pursued the goal of interactivity through real-time displays and the application of real-time systems techniques.

7.2.1 Communications

Currently only point-to-point communications are used but there is scope for the utilisation of reliable multicast once it becomes available. Although USS was not designed with a specific bandwidth in mind, it is clear from the results presented in chapter 6 that anything below 10 Mbps would be unsatisfactory due to the associated latencies. Two communication structures have been adopted by USS. Firstly, a client/server paradigm is used between processes within the same node, but the communication paths are heavily influenced by a hierarchical organisation, e.g. messages to other nodes are routed through the UM. Secondly, communication between nodes is strictly hierarchical.

7.2.2 Data and Computation Management

The method of monitoring state updates which are only sent by the owner when changes are made can be classified as passive partial replication. This technique is used between USS nodes but all data is replicated in each system, with only system-unique data being transmitted between them. Localisation, which also has implications for computation management, is supported through the use of constraint functions in the UM. Complementing the choices of data management is the complete distribution of computation between processes within a system. Rather than distribute computation between systems, it is completely replicated in every system. Process migration is supported, thus increasing the scheduling options and hopefully efficiency. As discussed in section 4.5.4.10, arguments can be made for the use of all 3 levels of behaviour distribution. Most of the systems reviewed supported the transmission of an entity’s state variables, whether continuously, by request, or only when a change of value has occurred. Level 1 distribution (commonly called dead-reckoning) was used exclusively by WAVES and DIS. Despite the potential display side-effects of this technique it is quite effective in reducing bandwidth consumption.
If necessary, it is possible to implement dead-reckoning with USS on top of the basic state management system.

<table>
<thead>
<tr>
<th>Feature</th>
<th>dVS</th>
<th>AVIARY</th>
<th>USS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Supported?</td>
<td>Real-time Supported?</td>
<td>Yes</td>
</tr>
<tr>
<td>Communications</td>
<td>Transport Mechanism(s)</td>
<td>Point-to-Point and Multicast</td>
<td>Point-to-Point (+ Reliable Multicast?)</td>
</tr>
<tr>
<td></td>
<td>Targeted Bandwidth Structure(s)</td>
<td>10 Mbps+</td>
<td>10 Mbps+</td>
</tr>
<tr>
<td></td>
<td>Client/Server</td>
<td>Client/Server</td>
<td>Client/Server &amp; Hierarchical</td>
</tr>
<tr>
<td>Data Management</td>
<td>Organisation</td>
<td>Passive Partial Replication</td>
<td>Complete Distribution</td>
</tr>
<tr>
<td></td>
<td>Localisation Support?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Computation Management</td>
<td>Organisation</td>
<td>Partial Distribution</td>
<td>Complete Distribution</td>
</tr>
<tr>
<td></td>
<td>Behaviour Level</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VE Modeling</td>
<td>Environment Management User Support</td>
<td>Parallel</td>
<td>Multiple</td>
</tr>
<tr>
<td></td>
<td>Multiple, Decoupled with Representation</td>
<td>Multiple, Decoupled</td>
<td>Multiple, Integrated or Decoupled</td>
</tr>
<tr>
<td>Time Management</td>
<td>Progression Method</td>
<td>None</td>
<td>Implicit</td>
</tr>
<tr>
<td></td>
<td>Node Synchronisation</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>Degree</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Security</td>
<td>Method(s)</td>
<td>None</td>
<td>Object Interface Level</td>
</tr>
<tr>
<td></td>
<td>Employed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1 Comparison of distributed VE feature classifications including USS.

7.2.2.1 Dead-Reckoning

Given an entity whose definition consists of a position vector and a velocity vector, a manager would monitor the velocity vector rather than the position. This would mean
that rather than sending continuous position updates as the entity moved, the manager could extrapolate a position from the velocity vector. If velocity was constantly changing then this technique would give little improvement. However, if two velocity vectors were maintained by the entity then even this can be accommodated. One vector would be used internally for the entity's own calculations and the second vector would be its exported property - monitored by the manager. The exported version would fuel its own approximated behaviour model (the same as the manager's) and updated only when its approximated behaviour differed significantly from its actual behaviour. This now replicates the same functionality that conventional dead-reckoning systems have.

7.2.2.2 High-Level Behaviour

Level 2 behaviour distribution can also be supported through another basic USS mechanism, that of remote UML function invocation. A number of functions would be defined to achieve some high-level tasks, such as driving around a corner, and then executed at the appropriate time. This technique can be used to control a user's shadow on a remote system (section 7.4.5).

7.2.3 VE Modeling

Multiple universes may be simulated simultaneously by a USS, although the prototype only supports one. There can be many users interacting in a simulation and within a system there is no special distinction made between an entity representing a user and an automated entity. Input devices are sampled from within the user entity, however, whether this is mapped into a direct device access or through a server process is an implementation decision.

7.2.4 Time Management

Both forms of time management are utilised by USS. Explicit time progression is almost a by-product when a distributed deadline scheduler is used to coordinate the simulation. To ensure synchronicity between individual systems, an implicit
progression model is used so that behavioural information generated by one system is valid in another.

7.2.5 Fault Tolerance

Fault tolerance is an expensive goal, best achieved by duplicating hardware and software components. However, there are a number of features of USS that lend themselves to at least a little reliability and recoverability - at a cost. The state held by a manager or entity may be reconstituted gradually through state updates or explicitly by request to the UM. If not enough information is held within a system to reconstruct the process, then it may be obtained from another system which is also simulating the same universe. If there is a problem with a particular node then entities can be migrated to another node. Alternatively, their state can be obtained from another system and started locally on another node.

7.2.6 Security

Security is another feature that can generally only be realised at a computational price. This aspect was not fully investigated because security measures can often hinder evaluation of other system features. However, there is basic access control support in that a process may locate the originator of any service request and the UML interpreter can limit access to OS services.

7.3 Important Features

The proposed architecture deals with a number of issues but there are a few aspects which are either worthy of note or unique to this solution.

7.3.1 Real-Time

A distributed real-time system forms the basis upon which the USS architecture is built. In order for the participants to efficiently interact with the environment and each other, it is important that they are provided with real-time displays. To keep in
step with the constant update rate of the displays, it is necessary to ensure that all entities are also updated at a constant rate. Failure to meet this hard deadline is a system failure. If all updates are guaranteed to happen within a given time frame it is possible to start accommodating for lags in the system by performing predictive calculations. When the simulation is distributed over a number of machines the network must also have deterministic properties if it is not to upset the processing deadlines. Predictability at this level also presents the opportunity to compensate for communications latency. It is likely, however, that determinism will be realised at the cost of performance and the under-utilisation of resources - a matter of concern to the designers of ATM switches where guaranteed bandwidth and bounded latency are primary requirements.

7.3.2 Scaleability

All of the distributed VE systems reviewed chose one mechanism for handling computation and one mechanism for handling data within the system. It is not possible to scale the system up or down without affecting the performance of such mechanisms. DIS, for example, replicates the data making up the VE on each node and partially replicates computation on each node through the use of dead-reckoning algorithms. Therefore if the VE has 10,000 entities, then each node must handle data and computation for each entity. Initially, when the number of entities in the simulation was in their low hundreds, this was not a problem. It was only when larger simulations were attempted that the idea of using localisation to reduce the workload of each node was suggested (section 2.3.3.1). In a similar vein, AVIARY uses a system model that works well when on a tightly-coupled network of workstations but will require some modifications if it is to support larger simulations. A similar story can be told for the other systems.

Adapting a design after the fact is always undesirable, because the end result is less attractive than it could have been if the design had taken a broader perspective to begin with. The architecture presented in this thesis is by no means perfect, but it does attempt to define a system that may be scaled from tightly-coupled
multiprocessors through to large scale networking of machines over large geographical distances.

The decision of when to network a machine as a USN in a larger USS or as a separate system requires further investigation. It is clear, though, that there comes a point when the network bandwidth between two clusters of machines can no longer handle the amount of traffic generated within a system. In order for users on either end of this connection to participate in the same simulation, two systems must be configured from these nodes that are capable of replicating each other's simulation workload.

7.3.3 Bandwidth Reduction

A great deal of effort has gone into reducing the amount of bandwidth used between processes and nodes, thus increasing the number of nodes it is practical to have in a system. Only those portions of an entity's state information that are of interest to managers are transmitted and only when a change in this information has occurred. Managers may also specify constraint functions that are applied to the state data the entities transmit to their UM. These functions can filter out unwanted data before it is sent to managers resulting in unnecessary computation and, more importantly, sent over lower bandwidth communication links to other nodes. Further savings could be made if a multicast protocol was available.

7.3.4 Modeling

The premise with which the process of VE modeling was approached in this thesis was that the development of VEs should not be constrained by past technological standards.

The need to model a VE is relatively new and is presently more of an art than a scientific practice. It is an exploratory process that often requires many changes before the model has reached a satisfactory state. UML is integrated into the USS architecture in such a way that the initial VE model can be developed off-line and then modified on-line. Any changes are reflected instantaneously throughout the
simulation. For example, a function describing the behaviour of an entity may be replaced by sending that entity a new UML definition for the relevant function(s). It is also possible to add or delete parts of the UML definition without affecting the existing state information for the rest of the definition.

The ability to build upon existing VE models is a powerful tool which can save time and cut development costs. Establishing a set of base environments with well defined core behaviours would ensure that VEs built by different designers would allow entities to move from one VE to another with reasonable ease. Although the movement of entities between universes was not implemented in the prototype, the ability to preserve those parts of an entity's state that are common in the source and destination universes is already in place.

Different perspectives on the same environment may be supported through the use of managers that monitor different components of the universe definition and display the contents in the desired manner. Alternatively, each manager may monitor the same information but only process those that meet certain criteria, probably with the aid of constraint functions to reduce bandwidth.

Any type of information may be modeled, subsequently the system has no knowledge of space *per se* and there is no requirement for it to be modeled. In fact, the UM and the core entity and manager libraries understand how information is structured, but do not expect any particular organisation, or look for any specific component in it. Consequently, only when a suitably dimensioned property (such as position) is added to the universe definition will space be modeled. Also, the relationship between simulation time and real clock time within the environment can be defined arbitrarily (section 4.5.5.1).

### 7.3.5 Flexibility

A minimal working system requires a RM, a UM and one or more ENT processes. In this state it is possible to run any non-interactive simulation. Although an entity may sample input devices, the user would not be able to see the consequences of their own
or the simulation's actions unless a manager was present, connected to a display. A manager may be introduced to monitor state changes and generate a suitable display. For VE simulations the two most commonly used managers would be VIS and AUR. However, non-interactive simulation may simply require a text-based display of key simulation variables. Managers are not only used for generating displays; for example, the SIM checks for violations of an entity’s space and informs the involved entities so they may resolve the situation.

The design of USS was driven by the desire to simulate interactive VEs, but due to its flexible structure it may be applied to other types of simulation. For example, artificial life simulators often use the model of a parallel processing, shared memory machine. Each “entity” within the simulation is a program whose instructions may mutate or, through breeding, become merged with another entity’s code resulting in a hybrid. This process continues over and over again. USS lends itself well to this problem because:

1. There is a direct comparison between the beings in the artificial life simulation and entities.

2. An entity’s code may be replaced at run-time and there is nothing to prevent the replacement code being generated by another entity.

3. In the same way, one entity may spawn another and define its behaviour through UML code generation.

The simulation would still operate within fixed deadlines but the update frequency could be reduced to sub-interactive rates. There may well be more efficient task-specific methods for the other types of simulation but USS at least provides a platform for testing ideas before developing the project further.
7.4 Areas for Investigation

A few improvements to the prototype were presented at the ends of chapters 5 and 6, but there are a number of areas encroached upon by distributed VE systems in general that the author feels need further attention.

7.4.1 Reliable Multicast

As a distributed system is scaled up, so the feasibility of using point-to-point communication links rapidly disappears. Multicast communications present the only practical solution: the overhead of a single transmission is incurred despite sending to multiple destinations. Unfortunately, the multicast systems that are becoming available now are, like their predecessors, unreliable. For data such as audio streams the occasional loss of a packet is acceptable. However, if state or event data is lost making its way from one machine to another then this will affect the state of the simulation. The consequences of this range from an event occurring on one node and not another, to users making a decision based on incorrect information. At the operating system level the consequences could be more severe, e.g. invalidation of a fault tolerance redundancy mechanism. Research into reliable multicast protocols is underway and the author believes that this work should be encouraged.

7.4.2 Guaranteed Bandwidth

Distribution over large areas not only increases communications latency between system components, but the latency also varies by greater amounts. Although it is impossible to totally eliminate latency, steps can be taken to account for it, but only if sensible estimates can be made. Fortunately, ATM permits the reservation of channels of fixed bandwidth between the communication's endpoints. Adoption of a technology that provides this kind of service at all levels, from LANs through to WANs, would also seem to be an essential component of future large-scale distributed VE systems.
7.4.3 Time Synchronisation

In order to synchronise time between machines there would appear to be two basic options: use a software protocol, such as NTP, or a satellite-based system such as SPS. If synchronisation over many hours is unacceptable, then the accuracy obtained using software protocols is quite low: within a few seconds. If simulation protocols can be developed that cope with this level of accuracy then this is sufficient. However, the author believes that the same amount of care given to estimating communications latency should be applied to that of time synchronisation. There is a solution available in the form of SPS which is currently prohibitively expensive (section 4.5.5.2) but, given a mass market and a little time, there is no reason why this technology would not become cheap enough to incorporate into every machine.

7.4.4 Real-Time Operating Systems

At the time of writing there are very few operating systems that can be used for real-time applications and are therefore expensive in comparison to the plethora of general-purpose operating systems. There are even fewer that support deadline scheduling and address the problems of distributed scheduling. The popularity of real-time systems research has risen somewhat since the widespread availability of multimedia workstations, but significantly more work is needed in this area before they may be effectively utilised for interactive real-time VE simulation.

7.4.5 Shadowing the User

There comes a point when latency is so great that simply reflecting every single change in a user's state to all other machines becomes impractical. A solution is to only transmit actions between machines and let the user's shadow processes effect these changes in the mirrored environments. This presents three problems that must be resolved. Firstly, how these actions are recognised; secondly, how they can be described in a form suitable for transmission and, thirdly, how these actions are interpreted. The first and last problems will be heavily influenced by the type of simulation in that the nature of the actions exhibited will vary. For example,
parameterised actions in a networked driving simulator may be reduced to accelerate, decelerate, turn left, turn right, navigate roundabout, park, etc. Whereas a Computer Supported Cooperative Work (CSCW) application might involve more intimate interactions between users. Consequently actions may even be required to mimic human gestures and facial expressions, e.g. approval, disapproval, happy, sad, etc. The format used for transmission of these actions may be as simple as executing a parameterised function remotely, or something more complex.

The choice of technique has implications for maintaining the integrity of the simulated environment. Consequently, more work is needed to assess the additional system functionality required to aid action recognition, representation and interpretation.

7.5 Outlook

The USS architecture has been dealt with in a rather isolated manner over the past few chapters. This section attempts to relate it to a few of the current popular topics in the area of VE systems.

7.5.1 Internet

The work in this thesis is not applicable to the Internet as it stands today: variable delays are experienced between communication endpoints and the available bandwidth may vary, to name but two problems. IPng (or IPv6) is essentially IPv4 (the current version) with some modifications (Bradner and Rankin, 1995). Aside from introducing techniques to reduce message fragmentation, preallocation of network resources is supported, allowing establishment of connections guaranteeing bandwidth and latency. Multicast has also been added as a standard addressing option for IP datagrams; in fact it has replaced broadcast as the base service abstraction, which is now a special case of multicast. Combined with a suitable transport mechanism from desk to desk, such as ATM, it should be possible to apply the USS architecture to the future Internet and certainly improve upon the prototype.
7.5.2 Virtual Reality Modeling Language

The Virtual Reality Modeling Language (VRML) is an attempt to bring interactive, 3D VEs to the Internet via the WWW (SGI, 1996). From a modeling standpoint, VRML is a classic example of a visual-centric approach. SGI's Open Inventor was chosen as the starting point for the format which, over the past two years, has been adapted to fit the role of a general format for describing VEs. After reconciling the representation of visual information with the need to model behaviour and the demands on the client browsers, it was decided to alter the way that the Open Inventor scene graph is used. This has been just one of many changes to the file format. Consequently, VRML has the same basic look as Open Inventor but is used in a different way. Audio has been added to the language and at the time of writing the more important problem of encapsulating behaviour is being addressed. Most people in the VRML community are agreed upon the fact that some form of programming language is required to describe object behaviour but no consensus has been reached on which language. The fact that this debate is happening at all reflects the problems of completely isolating information representation from simulation execution. It is exactly these problems that the USS architecture seeks to relieve through integrating the modeling process with the system that will execute the VE model.

7.5.3 Java

Java has been proposed as a language suitable for object behaviour representation within VRML. Java is interpreted, platform independent and increasing in popularity every day. Unfortunately for VRML browser writers, source code for a Java interpreter is not available requiring a lot more development work just to simulate a VRML scene. On the positive side, native translators are beginning to appear which greatly reduce the execution times of Java code. However, although Java can load classes (in byte-code form) at run-time, existing classes/functions cannot be redefined and there is no way of modifying data structures at run-time. Without these abilities, the VEs modeled using VRML will be very static in nature and require considerable amounts of time to develop and maintain. Specifically, if a VE is to be "upgraded"
then all users will have to disconnect whilst the new one is installed, possibly followed
by a conversion of old state data to the new format. Certainly not a quick or easy
procedure to schedule when the server is accessed by clients throughout the world.

7.5.4 Consequences

Some of the problems with the WWW and the Internet have already been described
(section 6.7.1). In addition to these, VRML is being developed incrementally from a
visual file format with the intention that it should one day also guide how machines
should be networked to realise interactive VEs over the Internet. By approaching the
problem in this way, the author believes that, in its current form, VRML will not fulfil
the expectations held by so many in the VRML community. For example, moving
entities from a VE served by one machine to a different VE served by another is not
possible unless some standardised structure for the information has been adhered to.
Currently this is not possible unless all the designers agree to conform to a given
structure and even then there is no way of enforcing such an agreement. A modeling
mechanism such as inheritance and a common set of base VEs would resolve this
problem.

The problems of distributed VEs are so many and varied that they must all be
addressed simultaneously to reach a well-rounded solution. Inevitably, however, the
lessons learnt by developing VRML will reinforce the validity of applying certain
techniques to distributed VE systems and may possibly even disprove others.

7.6 Summary

This thesis has attempted to fuse research in distributed systems, real-time systems,
modeling, languages and human-computer interaction into one system capable of
distributing real-time interactive simulations. Those issues examined (to varying
degrees) just within the area of distributed systems support were: message passing,
marshalling and unmarshalling, naming and name resolution, heterogeneous nodes,
scheduling, process migration, configuration management, performance management, time, synchronisation, security and persistence.

The problem domain is so complex that the exploration of the issues and their interdependencies within the time permitted was relatively limited. Many decisions had to be made during the design process, all of which were biased towards a system capable of supporting multi-user, interactive, VE simulations. Interactivity demanded a real-time system and multiple users required a distributed architecture with comprehensive techniques to maintain the integrity of the shared VE. Of the requirements presented in section 4.2, applicability was represented by the modeling language and its integration into the system, whilst fault tolerance and security took a back seat.

The architecture's structure is based upon the philosophy that the right tool is used for the right job. The combination of different distribution techniques, integrated with an expressive, flexible modeling language, has resulted in a scaleable system that can be used to both develop and simulate VEs in a heterogeneous, distributed computing environment.
Appendix A

UML Grammar

The grammar presented in Figure A.1 is for the data definition section of the UML. It is an extract from the actual yacc description used to implement the interpreter in the prototype.

/* Interpreter directives */
%token D_INSERT
%token D_REPLACE
%token D_DELETE

/* Keywords */
%token UNIVERSE
%token ELEMENT
%token PROPERTY
%token FUNCTION
%token VFUNCTION
%token CONVERT
%token CONSTANT
%token ENTITY

/* Primitive types and their literals */
%token INTEGER LINTEGER
%token REAL LREAL
%token STRING LSTRING
%token BOOLEAN

/* Name of a Universe, Element, variable, etc... */
%token <string> NAME
/* Comment */
%token <string> COMMENT

/* Constants */
%token C_FALSE
%token C_TRUE

/* Code constructs */
%token VAR FOR IF ELSE WITH EQUIV_OP

file : /* Nothing */
| file file_component

file_component: universe
| element
| constant_ext
| property_ext
| function_def
| entity_def
| D_INSERT
| D_REPLACE
| D_DELETE comp_ident dot_name

comp_ident: UNIVERSITY ELEMENT CONSTANT FUNCTION CONVERT PROPERTY ENTITY

universe: UNIVERSITY name_def '(' univ_body ')

univ_body: /* Nothing */
| univ_body univ_def

univ_def: constant
| element
| property
| converter
| function
constant_ext: CONSTANT ext_var_decl '=' initialiser ';' ;
constant: CONSTANT var_decl '=' initialiser ';' ;
initialiser: literal
 | '{' literal_list '}'
;
literal: LREAL
 | LINTEGER
 | LSTRING
 | boolean
;
literal_list: literal_list ' , ' literal
 | literal
;
boolean: C_FALSE
 | C_TRUE
;
element: element_decl
 | element_def
;
element_def: ELEMENT elemname '{' elem_def_body '}'
;
element_decl: ELEMENT elemname ' ; ';
elemname: dot_name ': ' NAME
 | dot_name
;
elem_def_body: /* Nothing */
 | elem_def_body elem_def
;
elem_def: constant
 | element
 | property
 | converter
 | function
;
property: PROPERTY var_decl ' ; ';
property_ext: PROPERTY ext_var_decl ' ; ';
converter: converter_def
 | converter_decl
;
converter_decl: CONVERT NAME ' ; ';

317
**Converter Definition**

```sh
converter_def: CONVERT NAME '({ }')

function: function_def
|   function_decl

function_def: FUNCTION function_proto ';
|   VFUNCTION function_proto ';

function_proto: dot_name return_type
|   dot_name param_list return_type

param_list: '()' 
|   '(' var_decl_list ')'

var_decl_list: var_decl_list ',' var_decl | var_decl

var_decl: NAME ':' type_decl

ext_var_decl: dot_name ':' type_decl

return_type: /* Nothing */
|   ': ' type_decl

code_block: '{' code_block_body '}'
code_block_body: /* Nothing */
|   code_block_body code_statement

code_statement: variable_decl

variable_decl: VAR var_decl ';

entity_def: ENTITY NAME ':' NAME '({ entity_def_body })'

entity_def_body: /* Nothing */
|   entity_def_body function

/* Name definition which may involve inheritance */
name_def: NAME ':' dot_name
```

318
```c
/* Type declaration */

type_decl:   primitive
           | primitive list_decl
           | NAME
           | NAME list_decl

primitive:   REAL
            | INTEGER
            | STRING
            | BOOLEAN

list_decl:   '[' INTEGER ']
             | '[' ']
```

Figure A.1 yacc description of UML grammar.
Appendix B

Benchmark Manager, Entity & UML Source Code

The simulation execution benchmarks described in section 6.5 used the minimalistic UML definition shown in Figure B.1. Those tests that monitored two components required the introduction of a second property: quanta2.

```plaintext
UNIVERSE Benchmark
{
    PROPERTY quanta : INTEGER;
    VFUNCTION Construct;
    VFUNCTION Update;
    VFUNCTION Destruct;
}

ENTITY bmark1 : Benchmark
{
    FUNCTION Construct;
    FUNCTION Update;
    FUNCTION Destruct;
}
```

Figure B.1 UML definition used in the prototype evaluation.

The source code for the benchmark entity is shown in Figure B.2 and the source code for the manager used to monitor the state updates is given in Figure B.3.
#include "ENT.h"

bool  EmbedFunctions( int noofParams, ... );
bool  Construct( void );
bool  Update( void );
bool  Destruct( void );

static ENT  *ent = NULL;
static UMLProperty  *quanta;
#ifdef TWO_PROPS
static UMLProperty  *quanta2;
#endif  // end TWO_PROPS
static UMLInstance  *quantaInst;

int main( int argc, char *argv[] )
{
    if ( argc != 3 || argv[1][0] != '-' )
    {
        cerr << "usage: " << argv[0] << " [-qt] entity_name\n";
        return ( 1 );
    }

    Process::IPC ipc = Process::IPC::IPC_NONE;

    if ( argv[1][1] == 'q' )
    {
        ipc |= Process::IPC::IPC_QNX;
    }
    if ( argv[1][1] == 't' )
    {
        ipc |= Process::IPC::IPC_TCPIP;
    }

    try
    {
        ent = new ENT( ipc, argv[2] );
        (void)ent->callback( MSG_UML_INIT_DEF, EmbedFunctions );
        ent->serviceEvents();
        delete ent;
    }
    catch ( ENTCTORError )
    {
        cerr << argv[0] << ": construction failed... terminating\n";
        return ( 1 );
    }
    catch ( ENTErr0r )
    {
        delete ent;
        cerr << argv[0] << ": terminating\n";
        return ( 1 );
    }

    return ( 0 );
}
bool EmbedFunctions( int noofParams, ... )
{
  UMLEntity *entity;
  if ( (entity = ent->definition()) == NULL )
  {
    cerr << "EmbedFunctions: can't locate entity definition\n";
    return ( false );
  }

  UMLFunction *construct, *destruct, *update;

  construct = entity->findFunction( "Construct" );
  destruct = entity->findFunction( "Destruct" );
  update = entity->findFunction( "Update" );

  if ( construct == NULL || destruct == NULL || update == NULL )
  {
    cerr << "EmbedFunctions: can't locate functions\n";
    return ( false );
  }

  construct->setCode( Construct );
  update->setCode( Update );
  destruct->setCode( Destruct );

  return ( true );
}

bool Construct( void )
{
  UMLCompType compType;
  UMLInstance::ID id = ent->instanceID();

  if ( (compType = ent->definition()->find( "quanta",
                                           (UMLComponent *&)quanta, id )) != UML_CPROPERTY )
  {
    cerr << "ERROR: can't locate property\n";
    return ( false );
  }

  if ( (quantainst = quanta->instance( id )) == NULL )
  {
    cerr << "ERROR: can't locate instance of property\n";
    return ( false );
  }

  #ifdef TWO_PROPS
  if ( (compType = ent->definition()->find( "quanta2",
                                           (UMLComponent *&)quanta2, id )) != UML_CPROPERTY )
  {
    cerr << "ERROR: can't locate property\n";
    return ( false );
  }
  #endif // end TWO_PROPS

  return ( true );
}
bool Update( void )
{
    static int count = 0;

    //
    // Don't perform any calculations, just mark the state as having
    // been modified.
    //
    quanta->modify();
    #ifdef TWO_PROPS
    quanta2->modify();
    #endif // end TWO_PROPS
    return ( true );
}

bool Destruct( void )
{
    return ( true );
}

#include <stdarg.h>
#include "Manager.h"

bool RegisterInterest( int noOfParams, ... );
bool Construct( int noOfParams, ... );
bool Destruct( int noOfParams, ... );
bool Update( int noOfParams, ... );

Manager *manager;

int main( int argc, char *argv[] )
{
    if ( argc != 2 || argv[1][0] != '-' )
    {
        cerr << "usage: " << argv[0] << " [-qt]\n";
        return ( 1 );
    }

    Process::IPC ipc = Process::IPC::IPC_NONE;
    if ( argv[1][1] == 'q' )
    {
        ipc |= Process::IPC::IPC_QNX;
    }
    if ( argv[1][1] == 't' )
    {
        ipc |= Process::IPC::IPC_TCPIP;
    }

    return true;
}

Figure B.2 Benchmark entity source code.

323
try
{
    manager = new Manager( ipc, argv[0] );

    manager->callback( MSG_UML_INIT_DEF, RegisterInterest );
    manager->callback( MSG_UML_CONSTRUCT, Construct );
    manager->callback( MSG_UML_DESTRUCT, Destruct );
    manager->callback( MSG_UML_UPDATE, Update );

    manager->serviceEvents();
}

catch ( ManagerCTORError )
{
    cerr << argv[0] << " : failed to construct... terminating\n";
    return ( 1 );
}

catch ( ManagerError )
{
    delete manager;
    cerr << argv[0] << " : terminating\n";
    return ( 1 );
}

return ( 0 );

bool RegisterInterest( int noofParams, ... )
{
    //
    // Identify which elements of the UML description we will want
    // to monitor.
    //
    manager->monitor( "Benchmark.quanta" );
    #ifdef TWO_PROPS
        manager->monitor( "Benchmark.quanta2" );
    #endif // end TWO_PROPS
    return ( true );
}

bool Construct( int noofParams, ... )
{
    va_list params;

    if ( noofParams != 2 )
    {
        cerr << "Construct: expecting 2 parameters for callback\n";
        return ( false );
    }

    va_start( params, noofParams );
    Manager::Monitor  *mon = va_arg( params, Manager::Monitor* );
    Manager::Entity    *ent = va_arg( params, Manager::Entity* );
    va_end( params );

    return ( true );
}
bool Destruct( int noofParams, ... )
{
    return ( true );
}

bool Update( int noofParams, ... )
{
    va_list params;
    if ( noofParams != 2 )
    {
        cerr << "Update: expecting 2 parameters for callback\n";
        return ( false );
    }
    va_start( params, noofParams );
    Manager::Monitor *mon = va_arg( params, Manager::Monitor* );
    Manager::Entity *ent = va_arg( params, Manager::Entity* );
    va_end( params );
    return ( true );
}

Figure B.3 Benchmark manager source code.
Appendix C

UML Benchmark Results

The charts for this appendix and UML source code and UML source code can be downloaded via anonymous ftp from:

ftp://ftp.dcs.ed.ac.uk/pub/rjh/uml
Appendix D

PML Benchmark Results

The charts for this appendix can be downloaded via anonymous ftp from:

ftp://ftp.dcs.ed.ac.uk/pub/rjh/pml
Appendix E

UM Benchmark Results

The charts for this appendix can be downloaded via anonymous ftp from:

ftp://ftp.dcs.ed.ac.uk/pub/rjh/um
### Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aural Manager (AUR)</td>
<td>Optional special manager that interfaces to sound generation hardware and provides a number of services to help manage the aural representations of entities. The nature of the hardware is irrelevant since the services provided are flexible enough to accommodate all forms of sound generation.</td>
</tr>
<tr>
<td>Computer Image Generator (CIG)</td>
<td>Special-purpose hardware that is dedicated to the task of generating three-dimensional graphics. A CIG may take the form of anything ranging from a stand-alone unit to a single printed-circuit board card. A CIG is connected to a general-purpose host that runs software controlling access to the CIG’s features (often in the form of a graphics library).</td>
</tr>
<tr>
<td>Console</td>
<td>Hybrid special manager and entity used for administrative purposes within a USS. It can be used to create/terminate system processes, introduce UML code into the simulation, etc.</td>
</tr>
<tr>
<td>Entity (ENT)</td>
<td>One of the essential system components. The state of the simulation is represented by the sum of each entity's state. Core ENT functionality is quite simple, primarily consisting of processing monitor requests, periodically updating its state, and sending state updates for those monitored components that have changed value. An entity’s functionality may be extended through the use of UML code.</td>
</tr>
<tr>
<td>Implementation Language (IL)</td>
<td>The language that the system processes have been implemented in, i.e. C++. This term is used to avoid confusion with the modeling language used, i.e. UML.</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Master USS (MUSS)</strong></td>
<td>The system in the prototype that manages the routing of messages from one (slave) system to another. It is also the first system to start in the network and is contacted by all other systems after they have finished their initialisation.</td>
</tr>
<tr>
<td><strong>Master UM (MUM)</strong></td>
<td><em>see</em> Universe Manager</td>
</tr>
<tr>
<td><strong>Node ID (NID)</strong></td>
<td>Unique identifier used to represent a USN within any given system. The prototype supports 32,768 nodes in one system.</td>
</tr>
<tr>
<td><strong>Process ID (PID)</strong></td>
<td>Unique identifier for a process within a USN and is used to contact and communicate with the specified process. The actual meaning of this identifier is implementation and node specific.</td>
</tr>
<tr>
<td><strong>Process Management Layer (PML)</strong></td>
<td>Software library used to abstract each operating system's differences to increase portability of the prototype USS. Services are currently restricted to message passing but could be extended to include time management, etc.</td>
</tr>
<tr>
<td><strong>Resource Manager (RM)</strong></td>
<td>Required manager that manages access to all resources on the node it is executing on. Contains a dynamic deadline scheduler to ensure that all processes complete their allocated workload on time, each simulation step. Aids the MUM in its system-wide load balancing duties by nominating processes that are consuming too many local resources.</td>
</tr>
<tr>
<td><strong>Resource Profile (RP)</strong></td>
<td>Data structure that can be used to hold details of the resource consumption of either an individual process or a node. An RP is the unit of communication between all processes when transferring information regarding resource usage. The resources monitored are: CPU, memory, backing storage and network bandwidth.</td>
</tr>
<tr>
<td><strong>Spatial Integrity Manager (SIM)</strong></td>
<td>Special manager that is used to monitor an entity's position and volumetric information. If one or more other entities should &quot;collide&quot; with each other, they are informed of the event and then left to resolve the situation amongst themselves.</td>
</tr>
<tr>
<td><strong>Special Manager</strong></td>
<td>All special managers are optional system components, but the usefulness of a system without them is limited. Often managers are used to control displays, e.g. VIS, but they may also simply provide essential services to entities, e.g. SIM.</td>
</tr>
<tr>
<td><strong>SUM</strong></td>
<td><em>see</em> Universe Manager</td>
</tr>
<tr>
<td><strong>System ID (SID)</strong></td>
<td>Unique identifier used to represent a USS within a network of systems. The prototype permits a maximum of 32,768 such systems to be networked together.</td>
</tr>
<tr>
<td><strong>Universal Configuration Language (UCL)</strong></td>
<td>Simple variable-value language with a hierarchical structure that is used to describe all configuration information within the prototype system.</td>
</tr>
<tr>
<td><strong>Universal Modeling Language (UML)</strong></td>
<td>The language used to describe the Virtual Environment to be simulated. Structured as one or more related universes, each containing a number of constants and properties that are used by the defined functions. Enables description of the entities inhabiting the simulation and their unique behaviour. UML code may be passed between processes at run-time and, through interpretation, issue service requests to managers, redefine an entity's behaviour, and so on.</td>
</tr>
<tr>
<td><strong>Universal Process IDentifier (UPID)</strong></td>
<td>Unique address of a process within all systems. This identifier is composed of a unique system ID, node ID and process ID. Two UPIDs are present in every message transmitted between processes, one detailing the sender and the other specifying the destination for the message.</td>
</tr>
<tr>
<td><strong>Universal Simulator Node (USN)</strong></td>
<td>The building block of a USS. A single USN can execute a complete simulation on its own but is commonly networked with other nodes to form a larger, more powerful system. The bandwidth available to processes is at its highest within a USN and latency is at its lowest. A USN is a user's gateway into the simulation of a Virtual Environment. Each node supports one UM, one RM, a number of entities, and zero or more special managers.</td>
</tr>
</tbody>
</table>
Universal Simulator System (USS)  
A USS is composed of one or more USNs and distributes the simulations amongst them in order to increase simulation speed, manage larger simulations, increase system fault tolerance, and permit multiple user interaction. One node in every system is nominated as a master and runs the MUM which manages communications with other systems. Each system within a network replicates the simulation workload and they keep each other informed of their users’ actions within the simulation.

Universe Manager (UM)  
The UM is the heart of each node. Most communications within a node pass through the UM. Those intended for other nodes are sent to the UM at the destination through the Master UM (MUM). The services that are most in demand are: progressing the local simulation; satisfying requests to monitor state information issued by managers; routing state information sent by entities; routing messages from local processes to remote nodes and vice versa.

The Master UM has the same responsibilities as a normal UM, but in addition it also manages system-wide scheduling (including the coordination of entity migrations from one node to another). Other special services include progressing the simulation within the system, coordinating communications with other systems, and controlling individual node activation and deactivation. All UMs on other nodes in the system are known as Slave UMs.

Visual Manager (VIS)  
Optional special manager that interfaces to a CIG and provides a number of services to help manage visual representations.
Bibliography


336


