New Perspectives on Architectures for Real-Time Mission Simulators: Agents, Ambassadors and Components

by

Malcolm John Corbin

A Doctoral Thesis

Submitted in part fulfilment of the requirements for the award of Doctor of Philosophy of the University of Surrey

September 1997

Supervisors:

Prof. Bernard Weiss & Prof. Michael Underhill

Department of Electronic and Electrical Engineering

University of Surrey.

Summary

Mission simulators are in widespread use for the evaluation of military systems under circumstances of appropriate realism. This thesis reports on a series of investigations into the architectures which could result from a synthesis of existing simulation methodologies with novel computing techniques being developed largely for the needs of the internet. The techniques selected were component architectures and mobile agent systems.

A framework for simulation based on a component architecture is presented. Entitled MulTiSIM, it permits models to be distributed over a network, and for their interactions to be unaffected by changes in physical distribution and model type. Instances of models can be assembled into arbitrarily complex distributed structures to permit modelling of complex entities, while a degree of structural transparency over interactions with such entities is maintained. Examples of real-time simulators developed using this framework are presented, including a driving simulator for the Thrust super-sonic car and a helicopter mission simulator at DERA.

The thesis goes on to suggest various generic roles for mobile agents in medium and large-scale simulations. These roles include communications management functions, dynamic control over model distribution and mediation of specific interactions. Prototypes of the communications management and mediation roles are described, the latter being implemented in the form of what has here been termed an ‘ambassador system’, employing only a sub-set of the facilities normally required for a system of fully autonomous mobile agents, while representing the specific interests of a simulation model within a remote operator’s station.
Responsibility for the thesis

I wish to acknowledge the guidance of my supervisors, Professors Bernard Weiss and Michael Underhill and Geoffrey Butler. This work, with its limitations, remains my responsibility and I would confirm that the experimental studies were conducted solely by me, except where collaborative work has been identified in the text, and that the writing of this thesis was also conducted solely by me.

M. J. Corbin
Acknowledgements and Dedication

I would like to express my gratitude to Professors Mike Underhill and Bernard Weiss and to Geoffrey Butler for their unstinting support and encouragement during the preparation of this thesis.

On the MulTiSIM framework, I would like to thank to Daran Crush for his insights into the workings of the JOUST simulator and for designing the section of software which passed information into Internet packets. On the THRUST simulator, Dave Roden wrote the Iris Performer display generator code, Steve Holter the instrument display and Phil Madge performed the integration of the final system.

On the HOVERS simulator, I would like to thank Peter Birkett who provided a view of the ultimate purpose of the simulator, as well as clarifying the physics of infra red transmission. Pat Johnstone and Phil Madge wrote many of the individual models, while Dave Roden and Trevor Manning designed the visual environment. The specific structural diagrams for HOVERS models were constructed in collaboration with Phil. I would also like to thank Prof. Ken Jackson of Ulster University for his helpful suggestions on the modelling methodology and finally Geoff Butler who had the original inspiration for this project.

I would like to thank Peter Sapaty and James Darling for an interesting collaboration reported in Chapter 5, and Peter Borst and Hans-Thomas Goest for their dedicated work in supporting the WAVE interpreter. Peter Sapaty’s ideas on distributed systems provided the underpinning philosophy for much of the work reported in Chapters 5-7.

I would like to thank Justin Mason for making his Java graph drawing package available on the Internet for incorporation into the experimental work reported in Chapter 8.

This work was performed with the support of the Defence Evaluation & Research Agency (DERA) and the Ministry of Defence.

Finally, I would like to dedicate this thesis to my wife Hilary, without whose encouragement I would never have embarked on a PhD in the first place.
Publications Resulting from this Work

Journal Papers


Conference Papers


Corbin M J, Birkett P R & Crush D F “MulTiSIM: a Distributed Object-Based Simulation environment in Ada”, European Simulation Symposium, ESS’93, Delft, October 1993, pp 405-410.


** Internal DERA Reports **


Corbin M J “Approaches to Architecture Design for Distributed Simulation”, DRA/AS/ASP/WP94140/1, October 1994, MPhil to PhD transition report.

Corbin M J "Potential Use of Agent-Based Techniques in Large-Scale Simulation",
DRA/AS/SID/CR96117/1, March 1996.

DRA/AS/SID/CR96122/1, March 1996.

Corbin M J “Applications of Mobile Code Techniques to the Monitoring of Distributed
Contents

Summary ...........................................................................................................................................i
Responsibility for the thesis ............................................................................................................ii
Acknowledgements and Dedication ...............................................................................................iii
Publications Resulting from this Work ...........................................................................................iv
    Journal Papers ........................................................................................................................iv
    Conference Papers ..................................................................................................................iv
    Internal DERA Reports............................................................................................................v
Contents..........................................................................................................................................vii
List of Tables ...................................................................................................................................xi
List of Figures ...............................................................................................................................xi
Chapter 1 Introduction ..................................................................................................................1
Chapter 2 Literature Survey ..........................................................................................................6
    2.1 Architectures for Reusability ..............................................................................................6
        2.1.1 Object-Oriented Systems ..........................................................................................8
        2.1.2 General Purpose Component Architectures ..........................................................11
        2.1.3 High Performance Component Systems ...............................................................12
    2.2 Distributed Mission Simulation .........................................................................................14
    2.3 Mobile Agents and Ambassadors ......................................................................................16
        2.3.1 Mobile Code Languages and Systems ......................................................................18
        2.3.2 Multi-Language Mobile Agent Systems ................................................................19
        2.3.3 Potential Applications of Agent Techniques ..........................................................21
        2.3.4 Summary ..................................................................................................................23
Chapter 3 A Framework for Distributed, Component-Based Modelling ....................................25
    3.1 Introduction .......................................................................................................................25
    3.2 Language Choice Considerations .......................................................................................26
    3.3 Main Requirements .............................................................................................................28
    3.4 The Multi-processor Object data-Base (MOB) ..................................................................29
        3.4.1 Structure of the Multi-Processor Object Base .........................................................32
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5. Communications Architecture</td>
<td>34</td>
</tr>
<tr>
<td>3.5.1 Generic Inter-Model Communications</td>
<td>35</td>
</tr>
<tr>
<td>3.5.2 Data Stores</td>
<td>38</td>
</tr>
<tr>
<td>3.5.3 Event Handlers</td>
<td>40</td>
</tr>
<tr>
<td>3.5.4 Inter-Processor Communications Protocol</td>
<td>41</td>
</tr>
<tr>
<td>3.6 Simulation Support Features</td>
<td>44</td>
</tr>
<tr>
<td>3.7 Description of the MulTiSIM Scheduler</td>
<td>48</td>
</tr>
<tr>
<td>3.7.1 Load Smoothing</td>
<td>49</td>
</tr>
<tr>
<td>3.7.2 Discrete-Event Scheduling</td>
<td>54</td>
</tr>
<tr>
<td>3.8 Discussion</td>
<td>56</td>
</tr>
<tr>
<td>Chapter 4 Examples of Simulations Based on the MulTiSIM Framework</td>
<td>59</td>
</tr>
<tr>
<td>4.1 The THRUST Super-Sonic Car Simulator</td>
<td>59</td>
</tr>
<tr>
<td>4.2 The HOVERS Helicopter Mission Simulator</td>
<td>60</td>
</tr>
<tr>
<td>4.2.1 Infra-Red Detection Modelling in HOVERS</td>
<td>62</td>
</tr>
<tr>
<td>4.2.2 Representation of Articulated Objects in HOVERS</td>
<td>63</td>
</tr>
<tr>
<td>4.2.3 Use of Protocols to Define Model Interactions in HOVERS</td>
<td>64</td>
</tr>
<tr>
<td>4.3 Conclusions</td>
<td>67</td>
</tr>
<tr>
<td>Chapter 5 Pilot Study on Model Mobility within Simulations</td>
<td>68</td>
</tr>
<tr>
<td>5.1 Purpose of the Study</td>
<td>68</td>
</tr>
<tr>
<td>5.2 Hybrid Distributed Simulation in WAVE</td>
<td>69</td>
</tr>
<tr>
<td>5.3 Description of the Mobile Models</td>
<td>74</td>
</tr>
<tr>
<td>5.4 Performance Models for Large-Scale Simulations</td>
<td>76</td>
</tr>
<tr>
<td>5.4.1 Model for Systems using Global Data Broadcasting</td>
<td>76</td>
</tr>
<tr>
<td>5.4.2 Model for Systems using Geographical Mapping</td>
<td>79</td>
</tr>
<tr>
<td>5.5 Discussion and Critique</td>
<td>82</td>
</tr>
<tr>
<td>5.6 Summary</td>
<td>85</td>
</tr>
<tr>
<td>Chapter 6 Generic Roles for Mobile Agents in Large-Scale Simulation</td>
<td>86</td>
</tr>
<tr>
<td>6.1 Background</td>
<td>86</td>
</tr>
<tr>
<td>6.2 Description of the Generic Roles</td>
<td>87</td>
</tr>
<tr>
<td>6.2.1 Mediation of Entity Interactions</td>
<td>89</td>
</tr>
<tr>
<td>6.2.2 Communications Management</td>
<td>92</td>
</tr>
</tbody>
</table>
6.2.3 Object Mobility ........................................................................................................... 97
6.2.4 Light-weight models .................................................................................................. 100
6.2.5 Environmental modelling .......................................................................................... 101
6.2.6 Situational Analysis ................................................................................................... 104

6.3 Example Application of the Generic Architecture - The Control of Aggregation .......... 105

6.4 Discussion .............................................................................................................................. 109

Chapter 7 Design and Prototyping Work on Generic Agent Roles ...................................... 111
7.1 Introduction ............................................................................................................................ 111
7.2 Design Aspects of Communications Management ............................................................. 111
    7.2.1 Efficiency ................................................................................................................... 111
    7.2.2 Scaleability ............................................................................................................... 112
    7.2.3 Generic Properties ...................................................................................................... 113
    7.2.4 Invariance to Network Topology ............................................................................... 113
    7.2.5 Fault tolerance .......................................................................................................... 114
    7.2.6 Load Balancing .......................................................................................................... 115
7.3 Design Aspects of Resource Management through Entity Mobility .................................... 118
    7.3.1 Stability ...................................................................................................................... 120
    7.3.2 Accommodating Component Structures .................................................................... 121
    7.3.3 Run-Time Resources .................................................................................................. 122
    7.3.4 Object State Transfer ................................................................................................. 124
    7.3.5 Scaleability ............................................................................................................... 126
    7.3.6 Fault tolerance .......................................................................................................... 126
    7.3.7 Light-weight Models .................................................................................................. 127
7.4 Prototyping of Design for Registration and Interest Agents .................................................. 128
    7.4.1 Prototyping: Methodology and Objectives ................................................................. 128
    7.4.2 Prototype Design ....................................................................................................... 130
7.5 Performance Model for Registration and Interest Agents .................................................... 136
7.6 Discussion & Critique .......................................................................................................... 142

Chapter 8 Description of the Operator Interaction Agent Prototype ........................................ 146
8.1 Introduction ............................................................................................................................ 146
8.2 Description Of Prototyping Work .......................................................................................... 148
List of Tables

3-1 Summary of Object Inter-relations within the MOB ............................................................32
3-2 Example of Load Smoothing Algorithm in Operation ..........................................................53
7-1 Main Interactions between Registration, Interest and Environmental Agents ....................116
7-2 Main Interactions of Site Search and Mobility Agents .......................................................117

List of Figures

3-1 Addressing Scheme for Data Base Entries and Object Attributes ........................................33
3-2 Structural Transparency Applied to Data-Store Access ......................................................39
3-3 Conceptual Structure of a Multi-Processor Simulation .......................................................47
3-4 Model Designer’s View of the Framework Environment .....................................................47
3-5 Main Components of Framework .........................................................................................48
3-6 Effectiveness of the Load Smoothing Algorithm .................................................................54
4-1 Model Components Comprising the THRUST-SSC Simulator ...........................................60
4-2a Model Interactions for a Tank Group Described in Terms of Protocols ...............................65
4-2b Definitions for the Protocols Used in Fig 4-2a ..................................................................66
5-1 Subdivision of the Simulation Arena into Discrete Spaces ..................................................70
5-2 Interactions between WAVE layer and models .................................................................72
5-3 Main Features of WAVE Programs Interacting with Models .............................................72
5-4 Update Rate for a Distributed Simulation with Global Broadcasting ...................................78
5-5 Update Rate for a Distributed Simulation with Broadcast Bus Saturation ...........................79
5-6 Update Rate for a Geographically Sorted Distributed Simulation .......................................81
5-7 Speedup for a Distributed Simulation with Broadcast Bus Saturation ..................................82
5-8 Speedup for a Geographically Sorted Distributed Simulation ...........................................83
6-1 Schematic of an Operator Interaction Agent in use ............................................................90
6-2 Schematic of the Use of Entity Interaction Agents ............................................................91
6-3 Geographical Mapping of a Server Network onto the Simulation World ............................94
6-4 Geographical Sorting by Registration Agent .......................................................................94
6-5 Discovery of Other Entities by Interest Agents .................................................................96
6-6 Interaction of Attention and Interest Agents for Long-Range Discovery ............................97
6-7 Site Search Agents for Dynamic Model Relocation ...........................................................100
7-1 Structure of WAVE program graph for a server node ......................................................133
Chapter One
Chapter 1 Introduction

Mission simulators are intended to immerse members of a human crew in a synthetic situation of sufficient complexity and realism that the responses and decisions which they make are in some way representative of their behaviour in similar real situations. This requirement leads to several generic characteristics possessed by the majority of mission simulations; one is that a significant number of models of a variety of different types will be used in constructing the synthetic situation, representing the various entities with which the crews are interacting; the second is that there will be considerable variation in the level of detail required of the representation of each entity, largely determined by the extent to which the human crews interact directly with them; the third is that the action should take place in real time.

The simulators of interest for the purposes of this thesis are those involving piloting tasks, in which complex vehicle dynamics must be simulated at a sufficient update rate to provide the impression of smooth motion. The combination of multiple entity dynamics simulations and generation of the accompanying graphical displays for different crew members at these rates produces a high demand for processing power. Currently this can only be met within a multi-processing system, typically distributed over either a local area or wide-area network, with separate graphics workstations used to generate the displays required. The wide-area solution has been used to connect together numbers of independent simulators to form a synthetic environment (Seidensticker, 1994).

The fact that typical mission simulators comprise a large number of different models interacting in a variety of ways leads to the difficulty of ensuring that existing and new models will inter-operate correctly when the modelling priorities and configuration required for a simulator changes. Simulators used for aircraft systems effectiveness trials are particularly affected by this, since the synthetic situation which they are required to model can change very significantly from one trial to the next. The primary aim of the work reported in this thesis was to design and, where possible, to prototype new structures for mission simulators with the objective of improving their flexibility in allowing existing software to be re-configured and re-used and new elements introduced with minimal modification to the old. The approach adopted was to seek a synergy between the structures and mechanisms to be found in existing distributed simulation systems and frameworks, and new ideas on software structure derived from different areas of computing.
Introduction

The two areas selected as showing most promise of allowing significant expansions in design freedom were component architectures, and mobile agent systems.

Component architectures were originally introduced for real-time distributed systems (Simpson & Jackson, 1979), and have more recently found applications to personal computing and Internet software (Sun, 1996a) where they allow tailored applications to be built by composition out of existing items. Mobile agent systems are under development, in particular for making better use of the communications bandwidth of the Internet by permitting delegation of decision making to software entities which are free to roam in search of the resources required to complete their set task. Chapter 2 contains a survey of these two techniques, together with information on the current state of distributed simulation systems for real-time use.

In Chapter 3 the first results of the first synthesis experiment are presented, the application of component architecture techniques to distributed simulation. This has resulted in the design of a framework (MultiiSIM) intended to support an object-based modelling paradigm in which the components, actually instances of object classes, can be formed into hierarchies to represent the structure of complex entities. These hierarchies can be extended indefinitely to increase the level of detail with which any particular entity is represented within the simulation, as well as being extended over the physical network by distribution to maintain processing performance. The communications scheme designed to support this paradigm is totally generic and transparent to the distribution of models. In addition, in order to simplify model design, it is responsive to the hierarchical structure, for reasons which will be discussed.

Chapter 4 describes some applications of this framework to real simulation problems, including a small-scale driving simulator for the THRUST Super-Sonic Car project, and a medium-scale helicopter mission simulator, HOVERS (Roden & Corbin, 1994), which operates over a network of up to 25 graphics work-stations and has been used for more than ten helicopter systems effectiveness trials within DERA, who sponsored this work.

Chapters 5 to 7 describe various schemes for the application of mobile agent techniques to distributed simulation. In Chapter 5 a pilot study is presented, performed in collaboration with other members of the University of Surrey, to look at the feasibility of constructing large-scale simulations purely from mobile entity models, self-organising at a local level, and hence requiring no global resources. This work employed continuous-time models with simplified
dynamics and limited-range sensing algorithms, which were nonetheless capable of exhibiting a range of behaviours. An idealised performance model was also developed to predict the speed and scaleability of this system, which was used to verify that it was indeed capable of being indefinitely extended over a suitable network topology, in contrast to the conventional approach adopted by the Distributed Interactive Simulation (DIS) standards (IEEE, 1994a,b,c).

Chapter 6 reports on an individual project to take this work further, introducing the constraints which are imposed by practical models of the types found on typical simulators, which may be of considerable size, may interface with a variety of data-bases and hardware functions and incorporate a variety of sensing and detection mechanisms whose characteristics determine the nature of the interactions which need to be modelled. The first two constraints mean that many of the entity models themselves cannot be mobile, while the last constraint requires that the detection of one entity by another take place over a wide and variable range of simulation space, dependant on the signature of the entity being detected, the sensitivity of the sensor, and various environmental effects.

The approach adopted to this problem was to define a set of generic roles which could be fulfilled by various specific mobile agents, from the interaction of which the functionality of the overall simulation system would emerge. The aspects of simulation management addressed by these roles included firstly the detection or discovery process referred to above, information from which could subsequently be used to create a minimal set of communication channels between entity models, secondly the problem of dynamic placement of those entity models which can still be considered mobile within the limited group of simulators which have the capability to run them, and finally the use of mobile agents to mediate in the interactions between parts of the simulation which must be closely-coupled, either between pairs of entities, or between an entity and a human operator. For example, the discovery process was envisaged as requiring a set of Registration agents, one per entity and responsive to the movements of that entity within the simulation space, which would perform a geographical sorting of all the entities, and which would periodically launch Attention and Interest agents to perform the actual discoveries, taking account of the signature and sensor sensitivity aspects of the operation respectively.

In Chapter 7 these roles were developed further to take account of some of the potential problems posed by their designs. These include the need for dynamic load balancing within the network
performing the discovery function to account for the uneven distribution of entities within the simulation space, the need to be able to guarantee the stability of the algorithms used to control the placement of mobile entity models within the simulation network, and the requirement that the operation of the whole simulation, and of individual models, within it be robust against the malfunction of one or more of the agents controlling it.

Also in Chapter 7 a prototype of a system of registration and interest agents is described, which was constructed using a mobile-code language, in order to establish the practicality of this type of approach. A performance and scaleability model of this system was also derived, as an elaboration of the model of the mobile entity system presented in Chapter 5. Results from benchmark tests on the prototype were used to estimate parameters of this model, and hence to predict the performance to be expected of full-scale systems of this type. As a result of both parts of this work, various conclusions could be drawn about the nature of the computing environment within which this approach to simulation management would stand the greatest chance of being successful.

Chapter 8 describes another application of a mobile code technique whose design includes elements derived from several parts of the preceding work. This is an enhancement to the capability of the distributed framework of Chapter 3, providing a remote diagnosis and control facility for instances of distributed models operating within the framework. This monitor can be tailored to provide special facilities for particular models, which are available for use on remote operators' stations, but without in any way compromising the component architecture of the framework by requiring any alterations to these remote stations. This is an example of what has been termed here an "Ambassador System", a sub-set of a mobile agent system, and was prototyped by means of a group of Java applets (Gosling & McGilton, 1995), which operate through a standard world-wide-web browser interface.

Finally, in the conclusions, the novel aspects of the various parts of the work reported in the thesis are drawn together, and various possible lines for further work are suggested, including investigation of the use of the new language features to be found in Ada95 and Java to simplify the structure of the MultISIM simulation framework of Chapter 3, and to open the possibility of using inheritance to derive new model classes for it, the need for a data-base to record the characteristics of model components and the nature of their interactions to ensure that assemblies
of these components operate as intended, and to generate the various configuration files required
to launch a simulation, and finally the possibility of extending the monitoring capability
described in Chapter 8 to provide a comprehensive browser-based control suite covering all
aspects of the operation of a mission simulator.
Chapter Two
Chapter 2 Literature Survey

This Chapter will review the state of practice in three apparently unrelated aspects of software systems design, namely reusable systems, distributed simulation and mobile agents. The benefits to be gained from possible syntheses between these areas will be explored in the remaining chapters of the thesis.

2.1 Architectures for Reusability

In this section consideration will be give to various approaches that have been made to the problem of designing large software systems so that they can be readily reconfigured and so that elements of them can be reused in different contexts. Discussions of re-usability have frequently taken as their starting point the “3C reference model” (Latour et al, 1990; Weide et al, 1990).

This model distinguishes:

- **Concept**: a statement of what a piece of software does, factoring out how it does it; abstract specification of functional behaviour.

- **Content**: a statement of how a piece of software achieves the behaviour defined in its concept; the code to implement a functional specification.

- **Context**: aspects of the software environment relevant to the definition of concept or content that are not explicitly part of the concept or content.

This model supports the idea that a re-useable component should have an abstract definition, which may be implemented by one or more concrete examples. Components wishing to use the capabilities of another do so by means of an interface only to the abstract definition. Weide et al (1990), following Booch (1987), provide a list of desirable characteristics of a reusable abstract component, which should:

- "be clear and understandable (clarity);

- state everything about the behaviour that is expected of a correct implementation and nothing more (restrictiveness);

- support a variety of implementations and especially not rule out efficient ones (generality);
export operations whose functionality is so basic it cannot be obtained by combinations of other exported operations, i.e. it should export primary operations (primitiveness);

not export operations that can be implemented using the primary operations, i.e. secondary operations, unless the component is an extension or enhancement of a more basic reusable component that exports the primary operations (potential completeness);

not depend on the behaviour of another abstract component for explanation of its functionality, unless it is an extension or enhancement of that component (low coupling);

encapsulate a single concept that cannot be further decomposed, e.g. a single type (high cohesion).

Most of the rest of the discussion of re-use in this chapter concentrates on the third of the ‘3Cs’, the contexts in which component systems can be constructed and the nature of the language and other support needed for this. Resulting systems will be compared with respect to three characteristics:

- **extensibility** - the ease with which additional functions can be added;

- **containment** - the extent to which changes to one component have consequences for the design of others;

- **composability** - the ease with which it is possible to make use of a component outside the context for which it was designed.

The first serious attempts at designing re-useable systems were closely related to the concepts of object-orientation, which will be covered in the next sub-section. The sub-section following describes various general-purpose component architectures which are currently finding favour for the greater degree of composability which they confer. Finally, some more specialised component-based systems, intended in the main for high-performance real-time and embedded systems, are described.
2.1.1 Object-Oriented Systems

The basic starting point for object-orientation was the idea of encapsulating data (the attributes) and the code which accesses them (the methods) together, binding them to form an object. Since the data became hidden within this object, any necessary changes to it would need only localised alterations to its methods. An extension of this idea was data abstraction, in which the object was regarded as being the definition of an abstract data type which could be then instantiated any number of times to produce a class of similar objects, each with identical behaviour but distinct life-histories.

The real power of the object-oriented (oo) approach begins to become apparent when the idea of inheritance is introduced. This allows new classes to be defined as specialised extensions of existing classes, adding new attributes and methods, and possibly overriding existing methods. A new sub-class is said to obey an is-a relationship with respect to its super-class, possessing all of its original properties, though possibly in a modified form, with some new ones of its own. This results in the construction of a set of polymorphic classes, having an identical external interface, but taking many forms internally. Finally the facility of dynamic binding means that any objects from this polymorphic set can be used interchangeably at run-time, since the decision as to which methods it would be appropriate to call can be made dynamically, depending on the nature of the object on which the call is being made. This is sometimes referred to as a message passing interface. More advanced oo languages sometimes support multiple inheritance, where a sub-class can inherit from more than one superclass. This can give rise to ambiguities in cases where the superclasses have methods or attributes with similar profiles, or are themselves derived from a common super-class (repeated inheritance).

There have been many object-oriented languages; the pioneers of object orientation were the designers of the Simula language (Dahl & Nygaard, 1966) for simulation. Smalltalk (Goldberg & Robson, 1983) took many of the ideas of Simula rather further, to the extent that everything in Smalltalk is an object, even numbers, and message passing is the dominant form of interaction between objects. This tended to result in early implementations of Smalltalk being rather inefficient. Objective C (Cox, 1986) developed oo extensions to C, with extensive class libraries to support the paradigm at run-time, while C++ (Stroustrup, 1986) also took C as a starting point,
but placed emphasis more on the compile-time capability to define classes, including multiple inheritance, and has little in the way of support at run-time. This has resulted in a very efficient implementation, and C++ has gone on to become probably the dominant oo language. This approach is not without its dangers, however, since the paucity of run-time support can make it difficult to debug obscure errors of program consistency.

The designer of Eiffel (Meyer, 1992) took the view that program safety should be a paramount requirement and defined an oo language with extensive consistency checks, including the ability for the user to define pre- and post-conditions for the correct operation of methods. A form of multiple inheritance is also provided. Consistency checking is also a notable feature of the Java language (Gosling & McGilton, 1995), in which the ambiguities of multiple inheritance have been abandoned in favour of a combination of single inheritance with multiple interfaces. This is a very powerful approach, since all objects possessing a particular type of interface can be treated as belonging to a single polymorphic type, even though they may not have been derived from the same super-class, and hence do not belong to the same inheritance tree. By contrast, in the majority of other oo languages, polymorphism, and hence the ability to perform dynamic binding, is restricted to the confines of a particular inheritance tree.

It is also possible to provide support for an object-oriented approach in more conventional languages, such as FORTRAN (Corbin & Butler, 1989) or Ada83 (Corbin & Butler, 1990a). This has been done in the form of run-time libraries which form a database recording the existence of objects and their interrelationships. Ada83 possessed some of the attributes of an oo language, providing encapsulation through its package structure, and data abstraction through the definition of new types. However, inheritance was only weakly supported and dynamic binding was absent. In Chapter 3 an Ada83 simulation framework will be described in which dynamic binding has been provided through a run-time object data-base, and is used extensively to control models and pass messages to them.

Object orientation in its pure form has proved to be only of limited use in promoting re-usability. Its main strength is in providing extensibility, through the definition of new sub-classes; in this way, the code of the super-classes is being re-used by the sub-class. However, in terms of the 3C model above, inheritance is being used to perform several functions, which may conflict. In
particular, it is being used to extend the concept, the abstract definition of the component, through specification inheritance, and simultaneously to extend the content through code inheritance. Weide et al (1990) cast doubts on whether code inheritance should be permitted at all, given the possibility that subtle errors may be introduced by manipulations of an object’s internal data by its inheritors.

Another problem is that inheritance may not be equally applicable in all problem domains. Since inheritance replaces complete methods, it is most easy to apply in cases where objects have many small methods, and good predictions can readily be made as to which ones are likely to need overriding in potential sub-classes. In the field of simulation, it is the author’s experience that such predictions are rather difficult to make - methods tend to be large, making use of extensive amounts of data, and it is very difficult to know in what areas the next set of modelling requirements will differ from earlier ones.

When considering other aspects of re-usability, object-orientation fairs less well. Sub-classes are somewhat fragile constructs, heavily dependent on their super-classes. Changes to the interface of a super-class are very poorly contained, requiring at least a re-compilation of all its sub-classes, and usually changes to the design of their source code in addition.

Composability, the re-use of objects in different contexts, is made difficult by another characteristic of many oo applications. In any but the simplest systems, objects need to make extensive reference to other objects in the system, and call upon their methods in carrying out their own duties. In this way the object classes in a typical application become highly inter-dependent and cannot be used with a different set of classes.

Some flexibility can be gained through polymorphism, since the objects being made use of can belong to classes which are polymorphic with the original; however, in most oo languages, this restricts them to belonging to the same inheritance tree. A good example of this type of structure is to be found in the Java input/output package, where input or output streams can be linked in any combination required to perform sequential data filtering operations.

As an alternative to inheritance, some languages have facilities for defining Generic packages of software, notably Ada. Such packages provide a definition of a set of general-purpose services
which can be parameterised to produce specific instances. This parameterisation can either be done by supplying specific data types to which the algorithms in the generic package can be applied, or by supplying specific methods to complete the definition of these algorithms, or by both techniques. As an example, Sreerama et al (1997) report an oo system in which performance tuning can be performed in an evolutionary manner using C++ generic templates.

Although at one time it was hoped that generic packages would solve the problem of software re-use, and allow the founding of a software component industry, this early promise has not been realised. Generic packages appear to have limited applicability, largely confined to cases where large numbers of very similar packages are needed. An example of this is given in Chapter 3, where the complete communications infrastructure of the distributed simulation framework is provided by instances of just two types of generic Ada package. Another limitation is that it is not usually possible to make direct use of a generic package in its generic form. In Ada83, code must be written to make explicit use of a specific instance of the package, forcing all such code to be compiled after the instantiation has occurred; there is no equivalent to the dynamic binding available in inheritance which allows new classes to be defined after the code which makes use of them.

The solution to these problems has been sought beyond object-orientation and genericity, by attempting to de-couple the elements of a system into completely independent components, an approach which will be considered in the following two sub-sections.

### 2.1.2 General Purpose Component Architectures

The main principle underlying component architectures is that a system should be broken down into interchangeable units which have no direct dependencies on each other. In particular, all communications between these components should be indirect, mediated by some supporting framework, so that units can be substituted for others with suitable functionality without in any way affecting the operation of the rest of the system. Many of the constraints which such systems must satisfy are described by Jackson & Boasson (1995). Systems of this sort are beginning to find their way into the general purpose desktop and internet computing areas.
One of the first such systems was **IntelligentPad** (Nagasaki & Tanaka, 1993), in which components, called Pads, could readily be assembled into larger applications. The underlying framework controlled the operation of the pads within an application, and performed the communication between them via typed ‘slots’. New applications could be very easily assembled from existing pads by making use of the system’s graphical application builder. Each pad had an iconic visual representation, and the operating connections between the different pads’ slots were made by positioning the icons for the pads. New pads could be defined using either an interpreted scripting language, or a Pad Definition Language which compiled to C++.

Systems with similar objectives have been produced by the major software houses, including Object Linking & Embedding from Microsoft, OpenDoc from Apple, IBM and others and a recent one which promises to have considerable influence, **Java Beans** from Sun (1996a). This takes as its foundation Sun’s object-oriented language Java (Sun, 1995), intended to provide completely portable computing over local and wide area networks (see Section 2.3.1).

In this system a component, known as a Bean, differs from a conventional object in having additional exploratory interfaces, through which an external framework can establish its operating requirements. This is intended to permit the development of the type of visual application editor as is provided with IntelligentPad, but this time as a third party facility. The beans also have run-time support for event handling between components, persistence and merging of the visual layouts of components within a composite application. Distributed applications can be built by making use of the industry-standard CORBA distributed object system (Kuhl et al 1994; Peck et al, 1995), or Java’s own Remote Method Invocation facility (Sun, 1996b), and bridges are provided for using beans as components within the other main component systems.

### 2.1.3 High Performance Component Systems

The component systems above were all designed for general purpose computing applications, where good graphical interfaces are more important than execution efficiency. Component architectures are also well suited to the construction of more specialised asynchronous and distributed systems in which greater emphasis is placed on performance.
The evolution of these systems can be traced back to MASCOT (Simpson & Jackson, 1979), in which components are scheduled independently by a kernel, and communicate only through dedicated links called 'pools' and 'channels'. A pool would deliver the latest set of values placed in it, while a channel would act as a first-in first-out buffer, delivering messages in the order in which they were queued and also providing a means for synchronising component executions. These basic communications mechanisms have evolved into a complete taxonomy of communications protocols, taking into account the influence of finite-length buffers and their interactions with the threads of control of the communicating components, which are proposed as a BSI standards for asynchronous communications (BSI, 1996).

More specialised component systems for particular domains have been demonstrated. Kaplan (1995) describes the Processing Graph Method (PGM), a graphical specification tool for specifying the data flows between distributed components, which has been applied to the analysis of sonar data using multiple instruction, multiple data (MIMD) parallel processing arrays. The use of PGM allows the application to be specified in a way which is independent of the hardware architecture, while still leading to highly efficient implementations.

Boasson (1993) describes SPLICE (Subscription Paradigm for the Logical Connection of Concurrent Engines) in which each component comprising an application interfaces only to an agent (not to be confused with the mobile agents in Section 2.3). The agents are responsible for dynamically forming the connections between components, based on a publish/subscribe mechanism in which different types of data have unique identifiers. The connections made can be local or over a network, without in any way effecting the nature of the components. This type of system does away with the need for an application building stage, and has no need of a separate distributed object system.

Browne et al (1994) have developed a component-based framework intended for the rapid prototyping of complex control systems, which can include the redundant management elements (voters) needed for highly failure-tolerant flight control systems. This uses commercial CASE tools to specify the topology of the system, combined with a library of tested components from which the final control system can be assembled using code generation. The approach allows...
rapid changes to the redundancy management aspects of a design to be made, by exchanging one type of voting component for another.

Chapter 3 will describe a component-based system specifically designed for distributed mission simulators, in which components can be formed into hierarchies to reflect the structure of complex entities being simulated, and in which the communications paths are entirely determined at run-time by algorithms written into the models, partially guided by the hierarchical structures which have been formed. But first, the state of the art in distributed mission simulation will be reviewed.

2.2 Distributed Mission Simulation

Improvements in high speed processing and specialised three-dimensional image-generators capable of presenting a reasonable approximation to the view from a moving vehicle in the 1980s, led to the construction of numbers of independent vehicle simulators, mainly intended for operator training. In the late 1980s it was recognised that there was considerable potential for operating these in concert to produce a “synthetic environment” in which many vehicles could be seen to be moving independently, and cooperating (or obstructing) in the conduct of a single military mission. The training value of this was thought to be significantly greater that that of a single simulator.

The first noteworthy distributed simulation intended to provide the illusion that a complete military mission was being undertaken was the SIMNET system (Garvey & Monday, 1989) which was supported by the US Army and was intended solely to represent the motions of ground vehicles. A number of armoured vehicle simulator stations, distributed over several sites in the US, could be linked, and each driver and commander provided with a real-time display showing the motions of the other vehicles involved in the combat. However, due to limitations in the protocols employed, these motions were of rather poor fidelity, and relatively few vehicles could be accommodated within an exercise.

JOUST (Roden & Harrhy, 1992) is a system developed within DERA for distributed simulation of fixed wing air missions. It operates over local area networks, using modified Unix graphics workstations as piloted stations, and is intended to present pilots with a sufficiently realistic
environment in which to carry out beyond-visual-range engagements involving highly manoeuvrable aircraft. It does this by ensuring that a high update rate can be maintained for both processing and communications between workstations, using an entity-level protocol known as DDP.

The next development of SIMNET was to design a system which could successfully represent the faster and more complex motions typical of sea and air vehicles, so that a complete battle comprising elements from all services could be simulated. This new system was called Distributed Interactive Simulation (DIS), and incorporated several enhancements over SIMNET. The communications were defined as an extensible set of Protocol Data Units (PDUs), the contents of which became IEEE standards (IEEE, 1994a,b,c). The most important of these, the entity state PDU, incorporated dead-reckoning techniques to improve the fidelity of transmitted motion without increasing communications bandwidth requirements. A simple model of each entity’s motion, based on Taylor expansion of its current velocity and acceleration, was run at both transmitting and receiving ends. An updated entity state was only sent when the error between the transmitter’s model and reality grew beyond pre-set limits, meanwhile the receivers’ models were generating a smoother version of its motion.

DIS was developed for several years, and provided the basis for several complex exercises involving several thousand entities (Rogers, 1995). However, the limitations inherent in its design began to become apparent at an early stage. The built-in dead-reckoning techniques lend a certain characteristic jerkiness to motions of remote vehicles, making DIS simulations instantly recognisable. This leads to DIS being of marginal utility for demanding piloting tasks, such as formation flying or close combat (Crush, Corbin et al, 1996).

Attempts to increase fidelity by reducing error thresholds or forcing an increase in update rates run into the other main weakness of DIS, its relatively low communications efficiency. The entity-state protocol defined includes a considerable quantity of data, enough in fact to form a complete picture of the entity, including its identity, type, affiliation, attachments and moving parts, as well as its position, attitude and rates of motion.

The need for this is a consequence of the lack within the DIS protocol of any concept of simulation state. There is no repository of object data to which updates can be referred; instead
each transmission needs to be sufficiently complete to stand on its own. This situation is made worse by the requirement for regular transmissions, every 5 seconds, by objects whose state has not changed, to remind the others that they are still there. The net result of all this is to limit the maximum number of entities which can be accommodated to less than 1000, unless various ad hoc filtering schemes are introduced (Bassiouni et al, 1995). The source of the limit is not just the excessive communications bandwidth demanded, but lies equally in the resulting processing load imposed by the need for each simulator to sift through all of the information broadcast globally by the other simulators. This gives rise to a square-law dependency of total processing requirement on overall simulation scale; one of the principal objectives of the work reported in Chapters 5, 6 and 7 is the avoidance of this square-law.

Other problems of DIS relate to the inflexibility of the communications protocols, which are defined to the nearest bit. There is a frequent need in particular simulation exercises to tailor the nature of the communications as to its content, accuracy, frequency etc., and currently this cannot be done without breaking out of the DIS standard. DIS is strictly an entity-based system, as are the others described in this section, and assumes that all of the data describing an entity can be accommodated within the PDU originating from a single processor. Flexibility in communications and the representation of more complex entities, which may be broken down into an arbitrary number of independent models which are themselves distributed over the network, are two of the objectives of the generic object-based framework which will be described in Chapter 3.

First, however, we will review the state of research into distributed systems involving mobile code, which will become relevant to the work reported in chapters 5 to 8.

2.3 Mobile Agents and Ambassadors

In this section a number of different approaches to the problem of constructing systems employing mobile code will be considered. Mobile agent systems are regarded as important for their ability to autonomously perform a number of tasks on behalf of an operator, moving across a network to the best location for performing each task to minimise communications bandwidth (Magedanz et al, 1996). This section will not attempt to be a comprehensive survey of the field,
but will rather illustrate the main lines of approach being adopted by the many groups working separately in the area.

Environments for the support of mobile agents are being developed largely because of the needs of the Internet. The traditional use of the internet has been to access static data from remote locations. More advanced uses are being developed which involve the need for remote operations, such as performing specialised searches on remote data-bases or interacting with sites over discontinuous communications links, such as those used to connect mobile personal digital assistants (Wayner, 1995). Some of the proposed applications of mobile agents will be discussed in Section 2.3.3 below. The first part of this section will give an overview of the main mobile agent systems under development.

**Definition:** At this point it would seem to be useful to attempt to define what is meant by the term 'mobile agent'. For the purposes of this thesis, we will define a 'mobile agent' system as one which operates by virtue of transferring entities which include executable code between the processors on a network, where the transfers are under the control of the entity, and preserve its execution state. For the purpose of this definition, executable code is held to include code for an interpreter.

In Chapter 8 it will also be found to be useful to invent the concept of an alternative and rather simpler type of entity, which will be referred to as an 'ambassador'. A system of ambassadors also operates by virtue of transferring executable code, but has much less freedom of movement. The transfers are initiated by other (usually non-mobile) entities, and execution state need not be preserved after the transfer. In addition, to be a true ambassador, this entity must in some sense represent the interests of another (remote) entity within the environment to which it has been dispatched, i.e. it is "an honest man sent to lie abroad for the good of his country" (Sir Henry Wooton, 1568-1639). One reason for the practical importance of this definition is that it admits the inclusion of systems constructed using relatively straightforward programming devices, such as Java applets and servlets (see below).

Both of these definitions exclude distributed object systems, such as the Common Object Request Broker Architecture, CORBA (Kuhl et al, 1994), which maintain records describing a system of objects distributed across a network and are able to transfer the state of an object
across the network by cloning it, and perform operations on remote objects by means of remote procedure call (RPC), a temporary transfer of relevant aspects of execution state. However, none of this involves run-time mobility of executable code.

2.3.1 Mobile Code Languages and Systems

One of the most advanced and well-established agent languages is the Telescript system from General Magic Inc. (White, 1994). This is an interpreted object-oriented language which supports autonomous mobile agents. It is based on a system of static ‘places’ occupied by dynamic agents programmed in Telescript. Agents may move to a place by the ‘go’ instruction, involving a ‘virtual ticket’ specifying the maximum duration and route, and initiate an interaction with another agent there by a ‘meet’ instruction, presenting a petition specifying the other agent and certain resource limits.

Agents not at the same place can still communicate via a ‘connection’, which can transmit Telescript objects (not agents) and any other information needed. Agents require cryptographic credentials before they can enter another organisation's region of control. They also have more temporary permits, specifying the instructions which can be used, life-time, size and processing cost in units called ‘Teleclicks’. The purpose of all these checks is to prevent agents from accessing forbidden areas in the host meeting places, or from making undue use of system resources there.

This is in contrast to the view taken by the designers of the Tacoma system (Tromso and Cornell Moving Agents, Johansen et al, 1995). Here the sole primitive operation is the ability to meet another agent at the current place. Mobility is implemented by meeting a special agent which saves current variable values and arranges for transportation of code and variables to a new location. The full execution state is not saved; the agent resumes execution from its start point. No provision is currently made for communications other than through mobility. Tacoma agents were initially written in Tcl (Ousterhout, 1993), but extensions to allow agents in other languages are being established.

IBM are also working on an autonomous mobile agent scheme similar to Telescript in some respects (Chess et al, 1995; Harrison et al, 1995). This has the idea of an Agent Meeting Point,
where agents can interact with each other, and with the server, similar to Telescript's place. They expect the introduction of itinerant agents to result in disintermediation, i.e. removal of brokers, resulting in direct interaction between clients and service providers. They propose several models of agent behaviour: a simple dispersal/retrieval model, a collaborative model, and a procurement model with many procurers, all based around providing network services to people.

The Java language (Gosling & McGilton, 1995) provides strong support for code mobility. The language itself was originally designed for implementing embedded systems for use in domestic appliances, and was designed to have an exceptional degree of portability. In its current implementation, Java programs are compiled into platform-independent byte-code which can be interpreted by a Java Virtual Machine (JVM) on any system which has appropriate support. Java has come to prominence with the appearance of 'Applets' - fragments of applications which can be downloaded from a web site and run within a web browser environment to provide dynamic and interactive effects within world-wide-web pages. Applets are an ideal tool for the implementation of ambassador systems, as will be demonstrated in Chapter 8.

Java comes equipped with a particularly extensive and well designed set of application programmer interfaces (APIs) for various standard network functions. A further extension of these is the provision of a 'Servlet' API, which will allow application fragments similar to applets to be uploaded to willing servers. These would then be executed remotely there, providing responses to requests made via their own private protocols. Servlets will provide a way of implementing complementary ambassadors to those available from applets. An applet ambassador will come at the request of the receiver, a servlet at the request of its owner.

Another characteristic of Java, which distinguishes it from the other interpreted languages used for code mobility, is that the byte-code has been designed as an intermediate step which could be compiled further to the native code of the platform on which it is to run. This would be done after migration, just before the agent starts execution, and promises to result in systems of greatly improved performance. Vendors such as Silicon Graphics have already released the first generation of these 'just-in-time' compilers.

The CyberAgent system (FTP, 1996) is a mobile agent system based on the use of Java. CyberAgents are provided with a 'travel plan' comprising a list of nodes which may be visited
and the mode of distribution, sequential or radial. Radial distribution is carried out by sending a clone of the agent to each node, where they execute in parallel, transmitting their results back to the origin in the form of Java objects. Sequential distribution allows a single agent to control the order in which it visits the nodes in its travel plan. Various levels of security authorisation are provided, ranging from simple password protection to full cryptographic signature verification, for the user to choose depending on the sensitivity of the application.

A CyberAgent can carry with it associated programs written in other languages, to be interpreted by appropriate facilities on remote nodes. The main limitation of the CyberAgents system is that it currently does not support interactions between agents, being mainly intended to support gathering of data from hosts and remote control of networks. The Aglet system from IBM Tokyo (Lange & Oshima, 1997), in contrast, provides extensive messaging facilities between its mobile agents, both point-to-point and multicast (a form of selective broadcast, relying on a publish and subscribe interaction model).

The WAVE language (Sapaty, 1988) is unusual in placing emphasis less on the security mechanisms required for agents moving across the internet, and instead concentrating on providing a set of complex navigational semantics through which the interactions of many separate mobile threads of program are mediated. In WAVE, programs execute on the nodes of a logical graph structure, whose nodes and arcs can carry arbitrary labellings. These labels, and the topology of the graph structure, which is dynamic, are created by program threads in the course of their operation and are used to convey problem-domain information between them. Program variables may also be left on nodes, or carried along with the program threads. No other forms of communication are provided, or are strictly-speaking needed.

As WAVE has been used in this thesis, and is rather unusual, a brief tutorial guide to the language has been included as Appendix A. The language was employed in the prototyping exercises described in Chapters 5 and 7. In Chapter 5 is described an application to a distributed simulation consisting entirely of mobile models, in which the individual threads of a WAVE program are each associated with one model. In Chapter 7 a more general application is covered, which comprises a system of mobile Registration and Interest agents. The intention of this is to accommodate models which cannot themselves be mobile, for reasons of size or access to
resources, and to allow the use of communications management algorithms based on the needs of a variety of simulated sensors, employing a geographical sorting principle, based on code mobility, to provide a scaleable system.

2.3.2 Multi-Language Mobile Agent Systems

The projects in the previous section were all centred on the use of one main language for the implementation of agent functions, though Tacoma is moving towards a multi-lingual capability. There is much to be said for the view that the support for agent mobility is orthogonal to the language needs of the agent designer, and that it should be possible to support several agent languages within a single mobility scheme.

This view has been embraced by the designers of the AgentTcl system (Gray, 1995), who have set out from the beginning to separate the agent server aspects of their system from language interpretation. Their core server keeps track of the status of currently running agents; authenticates incoming ones, and passes them to the correct interpreter; provides a hierarchical namespace for agents extending throughout the network; allow agents to send messages to each other, either as one-off events or by establishing two-way connections; and allows an agent to send itself or a copy of itself to another location. The complete execution state is preserved on transfer; agent persistence is also supported.

Finally, Agents for Remote Actions, ARA (Peine, 1996) is another project to provide a platform for mobile agent work which is language independent. The core platform would provide a common set of services for migration, control of security, interaction with other local agents through thread synchronisation, messages and semaphores, and access to other resources through the local operating system. It is anticipated that interpreters for a number of suitable languages would be incorporated into this core system; mention is made of Tcl and Java.

2.3.3 Potential Applications of Agent Techniques

Although most of the language designs above have concentrated on the perceived needs of the Internet, other potential application areas have also been receiving attention in the literature. These range from abstract problems in distributed spaces, close to artificial intelligence, to telecommunications and simulation.
One example of abstract problem solving is the work by Goss & Deneubourg (1991), who describe a simulation of a group of simple navigational robots employed to search an area without maps or central control, employing instead short-range interactions between the robots. Some robots behave as homing beacons for others, forming dynamic chains of beacons that thread through the area.

Assad & Packard (1991) combine mobile entities with genetic algorithms. They describe a computational model of organisms moving in an artificial ecology which trade with the others in their locality while searching for resources. If they survive long enough to reach maturity they breed, making up a new genetic mix by crossover of parent’s genes and by mutation. The successful organisms were found to evolve strategies which include forming mutually supportive groups.

In the telecommunications field, Appleby & Steward (1994) describe a hypothetical system of mobile agents designed to perform a traffic management and routing task in a large communications network. They place great emphasis on the robustness of the agent system. Having large numbers of agents ensures that a failure of a single agent will have little effect on the overall system; dynamic task allocation means that another agent can take over its responsibilities. They also hold that in a robust system there should be no direct inter-agent communication. In their system, communication occurs by changing the state of the underlying system, or leaving data in nodes, not by message passing between agents, in case failure of one agent could somehow propagate to others. The population of the agents is dynamic, and depends on the workload of the network. They propose a mechanism for measuring the overall number of active agents, enabling replacements to be automatically generated to maintain a given level of service.

Schoonderwoerd et al (1996) have taken a different approach to the problem addressed by Appleby & Steward. Their mobile agents are simpler, and communicate somewhat in the manner of trail-laying ants by building up deposits of virtual pheromone over the network. They are able to demonstrate reductions in congestion compared with the earlier scheme, and the elimination of artefacts such as circular routes to which it is prone. Christodoulou (1997) compares several alternative strategies for the reinforcement of successful pheromone trails.
Magenta (Mobile AGENTs for Telecommunication Applications) is a scheme being developed at the Technical University of Berlin (Magedanz, 1996) with the objective of providing an enhancement to the largely client-server based protocols currently proposed for use in telecom. systems. They plan to make use of a combination of CORBA and Java in a reference platform. The main emphasis of the project will be on the demonstration of agent-based subscriber services best summarised as “Intelligent services on demand”.

In the field of distributed simulation, relatively few proposals have appeared. Stone et al (1996) propose an efficiency improvement to the current DIS architecture which relies on mobile agents. A large part of the network traffic in DIS is taken up with state broadcasts from stationary entities, needed to avoid timeout. The Stone scheme has every stationary object sending a mobile agent to every other simulator, where it generates the required state information locally, until overridden by a genuine update from its owner. Unfortunately, this seems likely to increase processing loads far more than it would reduce communications loads.

Lubbers & Valentino (1996) in their A.G.E.N.T.S. programme (Software Agents for the General Evolution of Network Technology and Simulation) propose to investigate the application of Telescript to investigations of network latency in distributed simulations as a starting point to a more far-reaching programme.

2.3.4 Summary

It is probably premature to draw up any comprehensive taxonomy of features for the schemes outlined above, since most of them are still evolving, and all have to some degree to be regarded as experimental, with the possible exception of Telescript.

Some have extensive built-in security and validation schemes (e.g. Telescript, IBM Agents, Agent Tcl, CyberAgents), others regard this as an aspect which can be added later as a separate layer (e.g. Tacoma, Wave). Some systems rely on mobility as the sole means of interaction of agents across the network (e.g. Tacoma, Wave), while most others permit a mixture of mobility with more conventional message- or object-passing mechanisms.

Some schemes do not permit agents to interact with each other directly (e.g. CyberAgents, Wave, Appleby & Steward), limiting them to interaction with servers associated with nodes, while most
others treat interaction with other agents as the prime purpose of an agent’s behaviour, and include within that interaction with agents fixed to one node (e.g. Tacoma, Telescript, IBM Agents, AgentTcl).

Many designers recognised a need for a true multi-lingual capability, and provide their agent support capability as a separate API accessible from a number of agent languages (e.g. Agent Tcl, ARA and ultimately Tacoma), while others are more tied to a single agent language, with limited ability to call upon the use of other languages (e.g. Aglets, CyberAgent, Telescript, Wave).

The descriptions above have illustrated the considerable diversity of approaches being attempted by the various groups working on mobile agent techniques. This diversity derives partly from the different intended areas of application of each scheme, but one suspects rather more from the relative immaturity of the field. There is currently a dearth of fully working applications of mobile agents which could be used to demonstrate any clear superiority of one approach above the others. Chapter 7 of this thesis concludes with comments about the need for and value of various features in a mobile agent support system from the limited point of view of a single prototype application.

In conclusion, this chapter has covered three very dissimilar areas of systems design, component architectures, mobile agents and distributed simulation. The rest of the thesis is concerned with exploring some of the ways in which these can usefully be brought together to give systems of improved capability and flexibility, starting with the design of a generic component architecture for real-time distributed simulation in the next chapter.
Chapter Three
Chapter 3 A Framework for Distributed, Component-Based Modelling

3.1 Introduction

This chapter describes some of the considerations behind the design of a component-based framework intended to support the implementation of certain types of distributed real-time simulation. The simulations of interest for this work are ones which show a significant amount of variability in the interactions among the various software entities represented within the framework, that is, ones in which it is not possible \textit{a priori} to define localised limits for the interactions of any entity.

This characteristic is frequently met in airborne mission simulators, in which a number of entities move through an area of terrain, interacting with others in various ways, during the course of a simulated mission. It is not possible to say in advance which players will interact, or over what range the interaction will occur. Within the framework described here, this problem is tackled by broadcasting information globally to data-bases contained within every processor involved in the simulation. Models then scan these data-bases for the information they require.

Certain of these entities are controlled by human pilots, leading to a requirement for real-time operation; one of the most cost-effective ways of implementing such systems is to employ a group of graphics workstations, each of which can generate a different view of the engagement, linked by a medium-bandwidth local area network (Roden & Corbin, 1994). Many of the models needed within the simulation are likely to be legacy software, written in a variety of languages (C, Fortran etc.); there is a need to be able to use these within the same environment as models written specially for the purpose.

In attempting to design a software framework within which such distributed simulations can be conducted, the approach taken here has been to attempt to maximise the modularity or independence of the models. A component architecture has been adopted in which each software component can define a complete class of models, which can represent an indefinite number of separate instances, or objects, each with its own life-history. These different software components do not interact directly, since this would introduce undesirable inter-dependencies, but do so only through communications facilities provided by the framework.
The objects are employed to represent the major entities involved in the simulation, but such objects can also be used to represent constituent parts of these entities, increasing the level of detail with which an entity's behaviour is described. Equally well, an object could represent the behaviour of an aggregated group of entities, each one of which has only a rudimentary existence within the simulation. This gives the framework the ability to model a situation on multiple scales simultaneously within a single simulation environment, depending on the purpose that the each part of the simulation is being put to.

If the objective of real-time performance is to be achieved, then it is essential that the objects can be freely distributed among the different processors in the network, while retaining the ability to communicate with each other. This freedom is needed not only to ensure that it is possible to balance the computing loads between a heterogeneous set of processors, but also so that any special hardware constraints can be met. For example, when constructing crew-stations, specific inputs may only be obtained from equipment connected to certain processors and specific display screens must be used for certain views. This distributional freedom should be achieved without any changes being necessary to the models involved in a simulation.

A full description of MulTiSIM can be found in Corbin & Butler (1996). Its name was intended to convey the impression of a multi-model, multi-processor, multi-scale simulation capability and also to evoke the name of a previous framework, TSIM, intended for control system design work (Winter, Corbin & Murphy, 1983), which was in use in the UK aircraft industry in the late '70s and '80s and was distributed commercially by Cambridge Controls Ltd.

3.2 Language Choice Considerations

The Ada83 language was chosen for the main part of the MulTiSIM framework because of its high degree of standardisation, portability and good software engineering features. However, it is not fully object-oriented, as it lacked facilities for inheritance and dynamic binding. MulTiSIM makes use of Ada83's encapsulation and data abstraction capabilities to enable the definition of self-contained classes of objects with well-defined interfaces. An emulation of a particular model of Dynamic Binding is provided as a key part of the framework. This allows new classes of object to be introduced at a late stage and be called upon by the framework, without any need to recompile the core parts of the framework which control them.
The alternative object-oriented languages considered were: C++, Objective C, Smalltalk and Eiffel. C++ (Stroustrup, 1986) seemed to have inherited many of the loose and confusing constructs of its base language, C, while adding further possibilities for error in its object-oriented extensions. Objective C (Cox, 1986) and Smalltalk (Goldberg & Robson, 1983) were rejected on the grounds of relatively poor execution efficiency and lack of portability to platforms of interest. Eiffel (Meyer 1992), while possessing impressive language and software engineering features which would otherwise make it ideal for this application, did not possess the portability, the level of support or the widespread acceptance needed. At this time, Eiffel compilers were only available from a single company.

Of the languages which have emerged into widespread use since this choice was made, Java (Sun Microsystems, 1995) appears to show most promise. Its portability is superior to Ada's, being to a large extent guaranteed within the standard, and its run-time consistency checking is considerably more thorough than can be provided in an Ada environment, extending within the implementations of object methods and covering the handling of exceptions. There are still reservations about its run-time efficiency. Most Java implementations use interpretation of bytecode, which reduces performance by 5-20 times, though "just-in-time" compilers producing native code are soon to become available. Garbage-collection is another likely problem area for Java; not only does this take time, but it may occur at unpredictable intervals, causing erratic performance in real-time simulators. Careful design of the run-time support will be needed before Java can replace Ada in real-time applications such as this.

In designing this framework, no direct use has been made of inheritance to define new model classes in terms of pre-existing ones, partly because it was difficult to implement in Ada83, and also because it has been found that defining constituent objects can be a more flexible way of constructing complex objects for simulation purposes. For this reason, the term "component-based" has been used to describe the framework, in preference to "object-oriented".

The advent of Ada95 (Volz et al, 1994), with its single inheritance capability, will allow experiments to be made in the use of inheritance for defining hierarchies of model classes in Ada (or rather, Abstract Data Types, since the word ‘Class’ in Ada95 refers to an entire inheritance tree of types). One of the difficulties expected lies in ensuring a stable breakdown of functions...
between various parts of an inheritance tree, when the desired characteristics of all the models within it are not known in advance.

This is another area in which the features of Java would be well matched to this application. Java has a weak, but tightly controllable, version of multiple inheritance, best described as "single inheritance with extra interfaces". As will be seen below in Section 3.5.3, this would provide good support for the dynamic binding model used within MulTiSIM to accept events from elsewhere in the simulation.

3.3 Main Requirements

A conscious effort has been made throughout the design of MulTiSIM to maximise the independence between the designs of the different parts of the framework, and between the models operating within it. This is essential with any large assembly of software to minimise the knock-on effect of the changes which will inevitably be necessary. The architecture adopted for the models is essentially flat, with no direct compile-time dependencies of one model on another. In this respect, it has considerable similarity with the class of decoupled software architectures advocated by Jackson & Boasson (1995), another example of which is given in Boasson (1993).

The principal requirements for this framework were:

- That it should allow models written in a variety of languages to be imported from other environments.

- That models should be reusable in a variety of different simulation contexts.

- The interactions between models should, as far as possible, be independent of the class or type that they belong to, making it relatively straightforward to introduce new model classes without altering existing ones.

- The distribution of models between processors should not influence the operation of the models, simplifying the task of configuring a complete simulation.

- Both continuous-time and discrete-event models should be accommodated, as well as models which make use of both paradigms.

- Entities must be decomposable into a hierarchy of constituent parts, through any number of layers. Each constituent in a lower layer must be a full member of the object world, visible
to all other objects and able to communicate with them, so that the higher level objects in the hierarchy need not be concerned with its communications traffic.

- The framework should be portable to a variety of computer platforms, and allow the use of heterogeneous networks for simulation.

- Simulations constructed using the framework should be as economical as possible in their use of inter-processor communications, without at the same time imposing any additional processing load at the receivers in reconstructing missing information.

These requirements by themselves do not form sufficient basis for the design of a generic framework. The question of producing such a design is essentially a dual problem: on the one hand it is difficult to specify the framework facilities without knowing a lot about the models which will use them, and the nature of the support they require; on the other hand, it is impossible to design the models without having a specification of the run-time facilities they will make use of.

A prototyping approach has been adopted as a way of addressing this difficulty. MulTiSIM is actually the third in a series of schemes making use of Ada in an object-oriented fashion. In contrast to the approach advocated by Booch (1987), who advocated designs based on self-contained Ada packages, these schemes have all relied upon a central object data-base to provide a world of named objects. The first scheme was a single processor object database (Corbin & Butler 1990a,b), intended for general-purpose application in AI and simulation. This attempted to support inheritance, constituency and sub-sets of classes in a single package. In reaction to the complexity of this, the second scheme was a single-processor object data-base, intended for simulation, supporting only constituent parts (Corbin & Birkett, 1992) which used a dynamic binding emulation for both model control and all inter-model communications. In MulTiSIM, some of the basic ideas from this second scheme have been extended to a multi-processor environment, with entirely new mechanisms for the inter-model communications.

3.4 The Multi-processor Object data-Base (MOB)

The core part of the MulTiSIM framework is a data-base containing basic information about all the objects in existence within it. The information stored includes the object's name, references to its owner and to its class, and a list of constituent objects of which it is comprised. Each object
instance fits into a hierarchy of constituent parts, starting with a single “Top Object”. Class objects are also defined, owned by a single “Meta Object”.

The information in this data-base is replicated across all processors, so that each has a complete set of entries for all the objects in the other processors, as well as its own local objects. This replication ensures that access to the information in the data-base is fast, requiring no communication with other processors. When an object is created dynamically, an entry for it is made in the local processor's data-base, and a single message is broadcast to the other processors containing the information they need to create the corresponding entries within their own data-bases.

The principle that each basic operation on the data-base results in only a single message between processors is very important for real-time operation of a multi-processor system. The alternative, in which an operation would involve a request message, followed by a response message, results in increases in communications traffic and in unpredictable processor timings, neither of which are desirable for real-time operation. The single message principle has been followed throughout this work.

When applied to object creation, the single message principle means that a creation request made in one processor is a complete operation and returns a reference to the new object immediately, for use within the creating program. This is true even when the new object is an instance of a class implemented on another processor, and therefore does not yet have storage allocated to it. When an object is destroyed, the storage allocated to it is retained on a free-list, so that it can be reused when another similar object is created. This eliminates problems caused by attempting to use the garbage collection facilities provided by various compiler vendors.

As well as supporting objects which are instances of a class, the data-base also has support for "single objects", which are not associated with any particular class. A single object is used to refer to a package of software which has not been written in the object-oriented style, and only contains a single set of attributes. This is particularly important when reusing software from other projects which has not been written using data abstraction. Creation and destruction of single objects is handled rather differently from creation and destruction of instances, since there is no class object to refer to.
The final, and probably the most important, facility offered by the object data-base is support for an emulation of dynamic binding. Ada83 did not permit dynamic binding, which involves selective calling of object operations, dependent on the type of object encountered at run-time. However it is vital to have this ability, since it permits the construction of general purpose facilities which can call object operations without knowing at compile-time what classes of objects they will make use of. In other words, it allows the MulTiSIM framework to be written and compiled before any of the models which it will control are written. Ada95 would provide language-level support for dynamic binding (known as ‘dispatching’ in Ada) though only through a single inheritance tree.

The current binding method used in MulTiSIM uses Ada package bodies containing sets of Case statements which perform the dispatching function. These are produced automatically by a code generator immediately before linking. This code generator is itself written in a form of object-based Ada, and parses the Ada specifications of the models and of the messages to them generated by the framework in order to find matching operations. Matches result in generation of Ada code containing an operation body to be called by the framework which contains a Case statement with entries for each of the models that can perform that operation.

This process possesses several advantages for this application:

- The resultant code is standard Ada, and subject to all the compile-time consistency checks built in to that language. Additional run-time checks are also generated to guard against such things as attempts to call inapplicable operations on an object.

- Partial matches are permitted in certain circumstances, for example, models may omit unused parameters from operation calls, or have additional ones, as long as default values are defined for them.

- The overall result is to provide a facility similar in scope to multiple inheritance. Models may define arbitrary mixtures of operations from several sources, and be dynamically bound to all of them. This goes beyond the model of inheritance supported by Ada95 towards the multiple interfaces model provided in Java. In the context of the MulTiSIM framework, this capability allows a model to be bound to the operations provided by any variety of event handlers, in addition to those defined by the framework itself.
The MultiSIM framework makes use of dynamic binding to control the time integration of models, and to pass user-defined event messages to them from other models. The models are called through a standard interface made up of a set of 24 operations, described in Appendix B. This provides a relatively rich interface, which has proved sufficient to allow a wide variety of dynamic models to be defined, including continuous time, discrete-event and mixed models, both classes and single objects. Operations are provided to create, clone and destroy instances, to initialise and integrate continuous models, to initialise, start and stop discrete-event models and to store, restore and display the internal attributes of a model.

3.4.1 Structure of the Multi-Processor Object Base

Each entry in the MOB for an object is held in an array indexed by the object’s local identity variable (Id_Ref), and consists of the following:

- The type of object (instance, class, single object, Top_Obj or Meta_Obj);
- its global identity, unique within the simulation and recognised within each processor;
- its name, unique within the component parts of its owning object;
- a binding variable indicating which local software package contains its methods;
- the Id_Ref of its owning object;
- a list of its component parts;
- the Id_Ref of its class, if it is an instance;
- a list of its instances, if it is a class.

The relationships between the different types of objects, and the uses made of their parts and instances lists, are summarised in Table 3-1:

<table>
<thead>
<tr>
<th>Object type</th>
<th>Owner</th>
<th>Parts List</th>
<th>Instances List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance or single obj.</td>
<td>Top_Obj or real obj.</td>
<td>Other real objects</td>
<td>empty</td>
</tr>
<tr>
<td>Class</td>
<td>Meta_Obj</td>
<td>Deleted instances</td>
<td>Current instances</td>
</tr>
<tr>
<td>Top_Obj</td>
<td>Meta_Obj</td>
<td>Real objects</td>
<td>empty</td>
</tr>
<tr>
<td>Meta_Obj</td>
<td>Meta_Obj</td>
<td>Classes</td>
<td>Single objects</td>
</tr>
</tbody>
</table>

Table 3-1 Summary of Object Inter-relations within the MOB
The effect of this is that each currently existing real object occurs once and once only in the tree of component parts starting from Top_Obj, and once and once only in one of the instance lists found by searching the classes starting at Meta_Obj. The storage used by deleted instances is retained within each class and re-used when creating new instances to prevent memory leakage when creating and destroying objects.

The addressing scheme used to obtain access to the MOB and data store entries for an object, as well as its private attributes held within its class, is shown in Figure 3-1. The main factor influencing this design was the need for speed of execution, and so all the data relevant to one object can be obtained from a single index variable, with no need for any intermediate translation.
3.5. Communications Architecture

The framework has two independent sets of communications standards. The primary one is the inter-model standard, defined as a set of Ada procedure and function calls, while the second one is the inter-processor protocol. The main features of both standards will be described below.

The separation between these two standards is one of the principal design features of the framework. Many simulator designs, such as JOUST (Roden & Harrhy, 1992) or DIS (Seidensticker, 1994), have a single communications protocol for all purposes, which of necessity becomes a compromise between the conflicting requirements of the model designer and the communications engineer.

Separating the two interfaces has allowed each to be optimised for its purpose, with the following benefits:

- the inter-model interface has been structured as a data-base, removing problems due to the time-dependency of data reception from the concerns of the model designer.

- access to data through the inter-model interface is entirely independent of the physical distribution of models over the computing network, allowing the same model to be used in a variety of simulator configurations without change.

- the inter-processor communications can include aspects such as error-correction, synchronisation, simulator control and data selection and packing, which are not normal concerns of the model designer.

- it is possible to make considerable changes to the framework and inter-processor communications, for example to change from a real-time simulation system to a non-real-time analysis system, without in any way affecting the inter-model interface or the models.

However, one inevitable consequence of the adoption of this structure is that each processor making use of the inter-processor communications must be running a copy of the MultiSIM kernel in order to interpret them. This may not always be possible on some types of hardware platform, for example, controllers of motion bases. In this case, it is quite practicable to write a special-purpose gateway, running on a MultiSIM machine, to convert data to the format
required by the foreign system, and re-transmit it. This has been done to interface the HOVERS simulator to a helicopter motion base, the gateway being one of the set of models operating in the framework.

3.5.1 Generic Inter-Model Communications

The objects within this framework clearly need to communicate information governing the interactions between them. The framework has facilities to allow this communication to occur over a distributed world of objects, designed to satisfy the following requirements:

1) The nature of the communications should not be pre-determined by the framework, but must be under the control of the designer of a set of interacting models. Furthermore, it should be possible to add to the communications in an incremental manner, without invalidating what has already been defined.

2) The communications should be independent of model class. It should not be necessary to know what type of object is being communicated with either at compile time, or at runtime. This principle makes it possible to introduce new classes of object without redesigning, or even recompiling, the existing classes, provided that the nature of the communication does not change.

3) The communications must also be independent of the distribution of objects between the various processors in use for a particular job. This is vital if objects are to be reusable in different contexts. Precisely the same model code should be usable for a single processor simulation as for a multi-processor simulation.

4) A third type of independence is needed, termed here "structural transparency". In a compound entity, which is made up of a hierarchy of constituent parts, each constituent will be responsible for only a sub-set of the overall interactions of which the compound entity is capable. To preserve design independence between models, it is important for other objects to be able to communicate readily with the compound object without needing to know in detail how its operations are split between its various constituents.

5) The communications should permit scanning of all available information, and should consequently be optimised for situations where many more data accesses are being made than data updates. The purpose of this is to allow the writing of sensor models which
need to consider the totality of information available within a simulation when attempting to detect hitherto unknown entities.

The framework currently supports two distinct types of communication: one in which an object can request information about the state of another object, and a second in which one object can send a message to another. Both are intended for use on medium-bandwidth networks such as 10 Mbit/s Ethernet. Various architectural features of the communications are intended to meet the requirements above, as follows:

1) To ensure that the framework is truly general-purpose, the nature of the information to be communicated is determined solely by the designer of the objects. This lack of specialisation is achieved by providing the communication facilities in the form of generic Ada packages, which are instantiated with particular data structures as parameters to implement the specific communication requirements of the set of objects in use. The communications can be added to in an incremental manner simply by defining additional instantiations of these generic packages without in any way affecting earlier ones, or the models which make use of them.

2) Independence of model class is ensured by requiring that all references to objects, whether retrieving data from them, or scheduling events for them, take place using the referencing scheme defined by the MOB data-base, which is the same for all classes of object. Where necessary, it is possible to establish what class an object belongs to at runtime by obtaining a reference to its separate class object within the MOB.

3) Independence of distribution is guaranteed by similar means. All objects within a simulation are referenced by the same scheme, irrespective of their actual location within the processing network. Constituent parts of compound objects may be freely distributed between processors. Where necessary, it is possible to establish where a particular model is operating by obtaining a reference to its processor from the MOB. The communications mechanisms determine at run-time whether access is being made to a local or a remote object, and perform whatever network transmissions are necessary transparently to the models which are communicating. Data retrieval is in any case always a local operation, as described in paragraph 5 below.
4) The communications packages have features which provide a degree of structural transparency, in a framework in which each constituent is visible as a separate object. The solution adopted makes use of a form of attribute inheritance through the tree of constituent parts, so that information defined at a high level in a compound object automatically applies to all the lower levels, unless specifically overridden by a lower level constituent. A single object reference can then be used to obtain all information relevant to an object, irrespective of where that information is defined within the component tree. Similarly, the handling of discrete events can be arranged to take place at the level most appropriate within the constituency hierarchy, events originating at lower levels being referred upwards until a suitable handler is found. While not being a full solution to the structural transparency problem, these mechanisms do have the necessary characteristic of being very efficient in their implementation.

5) To simplify access to information published by models in a simulation, all the information of a certain type published by the models in a simulation is gathered together into a single data-base, combining inputs from all classes and all processors into a single array which can be scanned efficiently. This data-base is duplicated on all participating processors, ensuring that all data-access operations are local to the model requesting the data, and guaranteeing a high throughput for data access required by the many sensor models in a typical simulation. An update to this data is slower, but still reasonably efficient, since it is accomplished by a single entry in a message broadcast over the local area network to all processors.

Alternative object broker architectures have been developed which provide object interaction services independent of distribution (Kuhl et al, 1994; Peck et al, 1995). However, the current versions of these systems do impose overheads, partly due to the use of Remote Procedure Call (RPC) mechanisms for interaction. RPC requires a response from the originator of information and a pair of network messages whenever a request for information is generated, in contrast to the single message for update, and local access to information ensured by the MulTiSIM mechanisms.
3.5.2 Data Stores

Data Stores provide the means by which one object can request information about the state of another. They provide for the global information transfer referred to in the introduction. Data Stores behave like extensions to the object data-base; they can hold a data record of a certain type for each object, indexed by the object's identity. When information is placed in the data store in one processor, it is automatically broadcast to all the others, and thus becomes global data available for inspection by any object in the system.

The principle of structural transparency becomes important for data stores because of the way in which information is accessed by object constituent, rather than by entity. Most distributed simulations are entity-based, producing information in fixed formats which can be retrieved in their entirety once an entity's identity is known. Much of the flexibility of MultiSIM derives from storing information by constituent, allowing the amount of data describing an entity to expand as fresh levels of detail are added to its modelling representation. However, this does lead to a corresponding problem with the retrieval of this information, in that the object(s) needing the data will not, and should not have to, know in advance the details of the constituent structure of the entity which they are inspecting.

The data stores provide support for structural transparency by allowing automatic reference to a superior object's data by any of its constituents, if they do not have data of their own in a store. This means that data placed at a certain level in the constituent hierarchy can be interpreted as applying to all the levels below it, unless over-ridden by the constituent's own data. It is not necessary to know the level at which this data is defined in order to retrieve it. Figure 3-2 illustrates the operation of this feature.
Figure 3-2 Structural Transparency Applied to Data-Store Access

This facility is not a full solution to the problem, but when used in combination with suitable model designs, which place data at appropriate places in the constituent hierarchy, it has allowed a useful degree of structural independence to be achieved. A more comprehensive solution would involve performing a tree search through the constituent hierarchy to retrieve information. Facilities are provided to allow models to do this when necessary, but it has obvious run time penalties.

One feature of the data stores is that each one has an index to all the objects which have placed data in it. This can be used by an object retrieving the data to scan through all the data which is currently available, and thus explore the world of objects in which it finds itself, without needing to be told explicitly which objects it must interact with. This ability to form "blind connections" (Chervi & Sautereau, 1993) greatly enhances the flexibility of use of objects within the framework and the ease with which the object population can be modified.

Data stores also contain information about the time latency or staleness of the data within them to allow implementation of extrapolation algorithms to minimise errors due to latency. These algorithms are not an inherent part of the framework, since the choice of whether or not to use them is one of the design trade-offs best left to the object constructor.

The combined use of these data-stores and the object hierarchy can allow forms of data-access which add flexibility to a simulation. Structural transparency is one example of this, already described. Another example concerns the concept of the "specialist constituent", in which some specialised attributes of an object can be generated for it by one of its constituent parts but
positioned in the data-store as if they had been generated by the object itself. This allows functions of a complex entity to be broken down into independent and replaceable object models, while maintaining an invariant interface from the point of view of any other independent objects in the simulation. It encourages the re-use of existing models, allowing their capabilities to be extended by adding new attributes, without introducing the complex compile-time interdependencies that inheritance would bring. This concept bears some resemblance to the "attribute delegation" requirement included in the new DoD High Level Architecture for simulation (DMSO, 1995).

Another example of the combination of data-stores with object hierarchy is the possibility of combining information from different levels in the hierarchy to derive a resultant. The information at these levels would typically be coming from different processors, at different update rates, but the data-store mechanism provides a consistent mechanism for drawing it all together. An example of this principle applied to the position and attitude of articulated constituent parts is given below in Section 4.2.2.

The distribution of models across the processor network can be changed very straightforwardly at link time, since access to information via the object data-base and data stores, and the operation of the event handlers, is independent of the model distribution. However, it is possible to take advantage of the fact that inter-communicating models are in the same processor, by making the data stores used for their communication into 'local' ones, i.e. ones whose information is never transmitted over the network. In this way a considerable number of regular interactions between closely coupled models can be removed from the network. This hidden information can readily be made available, when required for logging, since a local data store can be converted to a global one by a single run-time command.

3.5.3 Event Handlers

In contrast to the data stores, in which the communication is initiated by the receiver of the information, an event handler allows the source of the information to initiate a communication. To do this, the source object schedules an event for the receiving object, together with some associated data, to occur at some specified time. This event is queued until that time comes, when the receiving object is called upon to execute a dedicated operation in response to the event.
Event handlers are instantiated by the object designer to handle sets of related events, each of which can have different parameters associated with it, and provide the means of constructing discrete-event simulations. Event handlers make use of the dynamic binding emulation facility to force objects to respond to events. This ensures that the event handlers can be defined independently of the objects which will communicate through them. Structural transparency is provided for by redirecting unhandled events up through the ownership tree until a level is reached at which the event can be handled - this could be thought of as similar to the Ada exception handling mechanism, but in a distributed context. Note that this redirection of events occurs at source and does not involve extra LAN transmissions - the single transmission principle still holds.

The dynamic binding mechanism used in MulTiSIM allows a model to respond to messages from any combination of event handlers - effectively permitting multiple inheritance of the operations defined by these handlers. Ada95 will only allow single inheritance, and hence will not be a complete replacement for the current MulTiSIM facility. The most promising line of development would seem to be use of the Ada95 mechanisms where they are applicable (e.g. when defining sub-classes of models and when responding to basic framework operations) and to retain the existing MulTiSIM dynamic binding mechanism for event handlers. Java, however, could directly support the MulTiSIM inheritance model, by defining each event handler’s messages as an interface class. Each model could implement any set of interfaces required to receive the events it needs.

3.5.4 Inter-Processor Communications Protocol

The basic inter-processor communications mechanism is the User Datagram Protocol (UDP) layer of the Internet protocols (Krol, 1994). This allows broadcasting of a data packet from one machine on an Ethernet to all others with a single transmission and with acceptably low overheads and time latency. The framework communications protocol has been implemented by specifying of the contents of the UDP packets.

The protocol used has a variety of different types of data relevant to the simulator, including:

- object creations and ownership changes,
- cloning of one object from another,
scheduler events,
- events for user-defined event handlers,
- changes in content of user-defined data-stores
- destruction of objects
- commands to set object parameters,
- simulator controls and commands.

The data items are transmitted in the order shown above.

Each of these data items can be described in a small packet of information for transmission. Transmitting each of these data-packets independently would present a large overhead, so they are assembled by the communications handler into larger Internet packets, and unpacked at the receiving end. Several such Internet packets may be sent by each machine if necessary.

**Error Recovery:** The UDP Internet protocol has error detection, but no error recovery built in. The most common source of errors in an Ethernet environment occurs when two processors start transmitting nearly simultaneously. Such clashes are detected by the Ethernet interfaces and cause loss of both messages. The likelihood of clashes occurring increases rapidly as the maximum capacity of the bus is approached, and means that it is not practicable to use more than about one quarter of the theoretical bus capacity.

Even respecting this limit, occasional message losses will occur, and the framework needs some mechanism for ensuring that the simulation in progress will not be unduly affected by them. To be useable within a real-time system this mechanism needs to operate with minimal time-latency, to impose only small communications overheads, and needs to operate in a way which is transparent to the model designer and to the rest of the framework, if possible. The sequence in which messages are received is significant, and must be preserved.

Various schemes were considered:

- multiple transmission of all data would have imposed an unacceptable overhead in bus loading and time.
a positive acknowledgement scheme, with re-transmission when no acknowledgement comes, would require every processor to acknowledge every broadcast. This would impose even worse penalties in bus loading for large simulations, as it scales as the square of the simulation size.

A system in which the receiver can request a re-transmission of missing messages has been successfully demonstrated elsewhere, but it was felt to impose too much latency for this application, as well as being complex to implement.

The scheme finally chosen involves segregating the data which is important to the consistency of the simulation from the less vital data, and employing multiple transmission just on the important data.

The vast majority of the data transmitted during a run comprises data-store updates, which are repeated at regular intervals - if one update is missed it has little effect on the simulation. The more vital data includes object creations and destructions and discrete-events, which must not be lost if the simulation is to remain consistent. Packets containing these data types are transmitted a number of times sufficient to virtually guarantee successful reception by all machines (three times in the current scheme).

Since the redundant messages follow each other immediately, there is minimal time latency for recovery following the loss of one of them, and message sequencing is preserved. The redundancy management software has been implemented as an independent layer within the communications packages, having little effect on the packages above and below it.

In practice, the impact on bus loading and execution times of these redundant transmissions is negligible since only a small proportion of data is affected, while the increase in reliability has proved to be more than adequate. Before the start of a run, all data is treated as more vital so that correct model initialisation can be guaranteed.

One other potential problem area is the restricted size of the buffers used to receive Internet data, which creates a risk that data could be lost were a buffer to overflow. To reduce this risk, the protocol provides for the use of multiple data buffers. Each machine transmits into a separate port which is associated with a separate reception buffer. This means that the number of buffers
into which data is received increases with the number of machines in the simulation, virtually eliminating the risk of data-loss due to buffer overflow.

Synchronisation data packets are transmitted using another, common, port and are used to force machines to wait before starting a new cycle of simulation until their fellows have all reached the end of the previous cycle. Currently, synchronisation is used for all machines before the start of a run, while the simulation objects are being initialised. Slaving to real time is optional during a run, certain machines such as image generators being allowed to free-run.

**Management of Communication Load:** The fact that models do not interface directly with the inter-processor communications, but obtain their data indirectly through data-stores, provides the opportunity to introduce various data-management schemes to minimise and smooth the data communications load on the inter-processor bus.

The scheme currently implemented is relatively straightforward, and involves transmitting data only when it has been freshly updated by a model. The data-stores maintain an internal list of data items which have changed since the last broadcast, and only these are put on the bus. In addition, if the model distribution allows it, a data-store can be declared local to a particular processor, entirely removing its traffic from the bus.

This scheme has worked well in practice, since it allows the model designer some control over the optimisation of data-bus loading. Other schemes could be introduced if bus loading became more of a problem. For example, for non-time-critical data it would be possible to arrange for only a portion of the fresh data to be transmitted in any one cycle, holding the rest over until the next one in order to smooth out peaks in bus loading. This need not affect the design of the models in any way.

### 3.6 Simulation Support Features

The MulTiSIM framework has a number of simulation support features in addition to the distributed object data-base and communications described above. These features allow for instance creation and initialisation as well as control over the running of a complete simulation.

Both continuous-time and discrete-event models can be accommodated. Indeed, the same model can have both continuous and discrete aspects to its behaviour. The operations of the continuous models are interleaved automatically with any discrete events so as to maintain them in time
synchronisation. Continuous models with different update rates can be accommodated within the same processor.

The control of both continuous models and discrete-events is performed by a scheduler, local to each processor, which has both real-time and non-real-time modes of operation. The schedulers on different processors can be operated in synchronism, or be left to free-run under control of local real-time clocks. In non-real-time mode, they decide on the length of the next time increment by a voting procedure based on broadcasts of next-event information. In real-time mode, a fixed time increment is used to give consistent performance.

The Scheduler is responsible for controlling the execution of all models within a processor. It supports two distinct types of execution: pseudo-continuous and discrete-event. Discrete-event operations are scheduled to occur at precise times in the future. In contrast, pseudo-continuous operations of models need to be performed at a certain minimum rate, but the precise time is not normally relevant.

It can be argued that all operations could adequately be performed using the discrete-event method. The framework design does not reflect this point of view, since the provision of a pseudo-continuous mechanism allows the scheduler certain flexibilities which are important in a real-time system:

- it presents the opportunity to spread model executions out over time so as to give the processor an even load.
- conversely, it allows model executions to be performed up to a particular time, when required by other events.
- if the processor is unable to keep up with real time, model update intervals can be selectively lengthened in an attempt to achieve it.

No individual model is in a position to control these aspects of execution management. These flexibilities only become readily available when control is vested in a scheduler able to review the totality of the model executions to be performed by its processor. Use of the discrete-event mechanism for this purpose would, by contrast, require each model to schedule regular executions for itself entirely in isolation from its brethren. A more detailed description of the scheduler, including its load smoothing feature, is given in section 3.7 below.
No one processor is in overall control of the simulation. Each is capable of broadcasting commands and events to influence all the others. An operator's interface linked in to any one of the processors can be used to create new instances anywhere on the network, or clone them from existing instances, to send commands to them to set up their initial conditions and control the simulation. A sub-set of these facilities is available to the models themselves.

A group of models making use of a common set of communications packages can be formed into a model archive, from which the models required for a specific simulation can be readily selected. These models should work together without needing any further modification. The development of archives of models for different purposes and levels of fidelity should greatly reduce the effort required to set up specific simulations.

Figure 3-3 shows the overall software/hardware structure for the MulTiSIM framework. Models can be written in a variety of computer languages, as long as each is provided with an Ada harness through which the framework can control the model. This feature is intended to encourage reuse of existing models written in C, Fortran or Pascal. Use of the generic communications mechanisms allows the interactions between different models to be specified in ways which do not depend on the mix of other models in the simulation, or on the way in which they are distributed between processors.

Figure 3-4 shows the model designer's view of the resulting computing environment in which each model executes. Note that there is no necessity for direct package-to-package communication between models, a fact which aids in distributing them flexibly.
Figure 3-3 Conceptual Structure of a Multi-Processor Simulation

- Arrow direction: control transfer
- Circle direction: data transfer

Figure 3-4 Model Designer’s View of the Framework Environment

Figure 3-5 illustrates the major software dependencies between the various parts of the framework and the models (the more fundamental software components are shown at the bottom.
of this diagram). This has been structured to ensure that the framework is independent of the models, and the models are independent of each other. In addition, they are all independent of the operator’s interface, which can therefore be readily exchanged for a different one. This feature considerably increases the portability of the framework, since most operating system dependencies will be confined to this interface. The current prototype interface is a straightforward keyboard handler, and has no system dependencies.

![Diagram showing the main components of the framework]

**Figure 3-5 Main Components of Framework**

The main application currently envisaged for this multi-processor framework is to real-time mission simulators. These comprise a number of "piloted workstations" - powerful graphics workstations equipped with a sub-set of aircraft controls - at which a pilot can command the operation of a single aircraft model within the simulation. A complete simulator comprises a number of such workstations, within which the aircraft models can interact with each other and with a variety of other active models, such as ground vehicles and installations.

### 3.7 Description of the MultiSIM Scheduler

The main purpose of the scheduler is to ensure that all discrete events are executed at, or close to, the time for which they were created, and all pseudo-continuous models are updated sufficiently regularly to given an impression of continuous motion. The strategy adopted for the continuous
models is for each one to be responsible for integrating its own equations of motion, using the method best suited to its dynamics. The MulTiSIM scheduler indicates to each model when it should perform an integration, and over what time interval, but does not impose any particular integration algorithm, leaving it up to the model designer to implement an integration scheme appropriate to the dynamics of the model. This makes it relatively straightforward to import models from elsewhere which contain their own embedded integration mechanisms.

The continuous models in HOVERS have very different requirements for update intervals, ranging over two orders of magnitude between the high speed dynamics of a missile interception needing to be performed at 0.01 sec, and the steering commands to a tracked vehicle, which can be performed at greater than 1 second intervals. The intention of the scheduler is to allow a random mixture of such models to be integrated on the same processor, without needing any human intervention to ensure smooth operation, using a load smoothing algorithm described below.

Another issue when operating models at different update rates concerns the information exchanges between them. For maximum accuracy, the information generated by the low-update models should be converted into a high update data-stream for input to the high-update models, as described by Haraldsdottir & Howe (1988), and as implemented in the DIS dead-reckoning algorithms. There are, however, many cases in manned simulation when this improvement in accuracy would not produce any perceptible improvement in realism, and the extra computing load could not be justified. The strategy adopted within MulTiSIM is not to provide any specific interpolation or extrapolation facilities for performing this function, but to make available the information needed by model designers who may find it necessary to do this in their application.

### 3.7.1 Load Smoothing

The load-smoothing problem for a real-time simulation becomes apparent when a processor is required to operate a number of models which require different update rates. A regular succession of high rate model executions must be periodically interrupted by low rate executions. The problem is to ensure that, as far as possible, a constant interval is maintained between the
high-rate executions to give an illusion of continuous motion, despite these interruptions. Although much work has been done in the area of continuous simulation, there appears to have been little published specifically on load smoothing for a mixture of continuous-time models, hence the subject was investigated, ultimately resulting in the algorithm described below.

The classical method of achieving this is through multi-tasking and pre-emptive scheduling, with the low update-rate models run as background tasks, interrupted in their execution by the high-update models operating at higher priority. Although widely and successfully used in computer operating systems for load sharing between different users, for the real-time simulator application it suffers from several problems. The major one of these is caused by the large number of context switches needed to run with several models at high update rates - the overheads involved in these rapidly degrade processor performance to an unusable level. Other problems concern the difficulty in controlling the operating system's scheduler to ensure that all models are given a fair share of the processor over short periods of time, and the difficulty of ensuring portability.

Some of these difficulties could be overcome by using a special-purpose real-time operating system instead of Unix. This would not improve portability, however. Instead, we have based the framework on a single process running under Unix, and constructed a scheduler within this process designed to fairly allocate execution times among models running at a wide variety of update rates, while attempting to spread the processor load out as evenly as possible. It does this by dynamically assigning models to execution slots, and letting each model update itself without interruption within its slot before starting the next model.

An optimal load-smoothing algorithm, one able to cope with a mixture of model types, would need to know the execution times of each model, so that it could minimise the length of the worst-case slot time. We have chosen instead to adopt a more straightforward sub-optimal algorithm which does not rely on knowing execution times. There are three main reasons for this choice:
The execution time of any one model varies considerably depending on the population of other models in the simulation, since each model spends a portion of its time inspecting data from the others,

there are considerable operational benefits in having a scheduler which can operate transparently without needing any information from the operator or model designer, other than the required update rate to be aimed at.

the algorithm used must be capable of re-scheduling model executions very rapidly whenever the set of models in use changes during a run, i.e. when a new model starts operating, or an existing one is destroyed. For this reason, any algorithm relying on lengthy optimisations would not be suitable.

The algorithm is as follows:

Given that the time interval between the scheduler's execution slots is $T_s$ seconds, and the update interval for a class of models is $T_m$, define the execution interval for the models to be:

$$E_m = \frac{T_m}{T_s} \text{ slots (rounded downwards)}.$$

For each model in the class, assign it to an initial slot number in the range $0 \ldots E_m-1$, starting with the first model in slot number $E_m/2$ (wrapping round from $E_m-1$ to $0$). Then continue through any other model classes having the same value of $E_m$, and assign their models to slots, continuing on from the last slot assigned to the previous class.

If $T_m < T_s$, then put $E_m = 1$, which will cause the model to be assigned to all slots, and execute the model a number of times within each slot equal to $T_s / T_m$ (rounded upwards).

This algorithm will accommodate models whose update intervals are longer or shorter than the basic scheduler interval. It relies on the assumption that models from the same class will take similar times to execute, and so should be assigned as far as possible to separate slots. In order to be able to make use of this assumption, all models from the same class must have the same
update interval, a restriction which is deliberately imposed within the model class interface (Appendix B).

Models from different classes are more likely to end up sharing a slot, but this can be shown to give a shorter worst case slot time than having two from the same class. Given two classes of models with different execution times, \( T_1, T_2 \) then the worst case is:

\[
T_1 + T_2 < \max(2 \cdot T_1, 2 \cdot T_2)
\]

irrespective of the relative sizes of \( T_1 \) and \( T_2 \) (this assignment would be the same as that produced by an optimal scheduler provided that \( T_1 < 2 \cdot T_2 \) and \( T_2 < 2 \cdot T_1 \) when there are no more than two models in each slot).

Table 3-2 shows a typical set of model executions distributed by means of this algorithm, where models with several different update rates have been included. The visual impression is of a reasonably effective spread of execution load over time.

A numerical measure of the effectiveness of the load smoothing algorithm has been developed, based on the mean difference in load between the fullest and emptiest execution slots, averaged over a wide range of model population sizes. Figure 3-6 shows how this measure varies as the smoothing algorithm is altered. In this case the starting slot number for models with update \( E_m \) is being altered from 0 up to \( E_m-1 \). Optimum smoothness, given by the lowest values of the effectiveness measure, is given by starting values around \( E_m/2 \) (the actual optimum for these curves is \( E_m \cdot 6/13 \)). The different curves on the graph represent sets of models with different ranges of \( E_m \) values.

One limitation of the scheme described above is that models which require more than one slot time to execute cannot readily be accommodated, since no provision is made to interrupt their processing. This is not a problem with most of the dynamics models found in simulators, which typically execute in a few milliseconds, but becomes an issue when more complex models, perhaps making use of artificial intelligence techniques, are considered. One way of permitting such models to run within the framework is to install them on a separate processor where the slot time can be made longer - of the order of a few seconds if necessary.
<table>
<thead>
<tr>
<th>interval</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>15</th>
<th>20</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>models</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>slot nos.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total in slot</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3-2  Example of Load Smoothing Algorithm in Operation
3.7.2 Discrete-Event Scheduling

As well as executing continuous models, the scheduler is also responsible for maintaining a queue of pending discrete-events, and making its models execute these at the appropriate time. The event queue is defined in a separate generic Ada package, which is instantiated within the scheduler to cover those types of events allowed within the framework.

User-defined events are scheduled via event-handler packages. These are instantiations of a generic event-handler. They operate by maintaining their own internal queue of pending events in each processor, including with each event any user-defined parameters which are necessary for it. The first event in this queue is represented by an event on the central scheduler’s queue in that processor. When the central scheduler reaches the point in time when this event is due, it makes a call to the appropriate handler package, which fetches the first event from its own queue, decodes it, and calls the model to execute it, passing any parameters with the call.
The executions of the pseudo-continuous models have to be interleaved between these discrete-events so that the information from them is up to date when the discrete-event happens. The framework has two distinct strategies for doing this, depending on whether or not real-time operation is required.

**Non-Real-Time Mode:** The non-real-time strategy trades speed for improved accuracy. Each discrete-event is allowed to occur close to the time for which it was scheduled, even though this may be very close to the preceding event. The next execution slot for continuous models is shortened to cover the interval, and to complete the integration of at least the high update rate models to the time of the next event.

In a multi-processor simulation, each processor will have a different set of pending events on its queue for execution by its local models. A mechanism is needed to ensure that the processors move forward in time by the same amount to the nearest event. A voting arrangement has been implemented in the communications package whereby each scheduler broadcasts the time of its next event to all others, and then chooses the shortest time interval out of all of these.

In this non-real-time mode, the performance of the framework will vary in an unpredictable way as the time intervals between discrete-events vary. This could result in some very short execution steps where events are very close together. The framework has two mechanisms for avoiding arbitrarily short time steps, which would otherwise lead to large truncation errors in model integrations, and excessive slow-down.

The first is the use of a fixed-point time-base for the scheduler. The event queue has a basic time granularity within which two events are assumed to be simultaneous. This is a constant defined in the fixed-time package, and chosen to be a reasonably short interval for the current style of simulation. This could be readily altered to adapt the framework for other purposes, for example, a high bandwidth engineering simulation would require a shorter interval.

The second mechanism is the definition of an update-rate tolerance factor within each continuous model. The integration step for a model is not shortened by more than this factor - if such a
shortening is requested, then the integration step is omitted, and the interval added on to the next step for that model.

It should be noted that the mechanism described above is not as efficient as those developed for pure discrete-event simulation using conservative algorithms (Chandy & Misra, 1979), but has been adopted to ensure that the continuous models on the various processors continue to operate in a synchronised fashion.

**Real-Time Mode:** The framework's real-time strategy for mixed continuous/discrete-event simulation is rather different, and allows some compromise in accuracy in return for more predictable performance in terms of update rates.

In the real-time mode, the basic update interval times of the processors remain constant. Any discrete-events which do not naturally fall on the boundaries of one of these intervals are artificially delayed until they do. In practice, this means delaying an event for typically no more than 40msec. This characteristic must be borne in mind when designing any discrete-event models whose operation depends on accurate timing, such as weapon aiming calculations.

Situations can arise when a processor is not able to keep up with real time at the update rates specified for its models. In these circumstances, the scheduler will lengthen the update intervals of the models in an attempt to maintain real-time execution. The maximum extent of this lengthening is given by each model's time tolerance factor, mentioned above, which must therefore be set by the model designer to a value which will not unduly compromise the stability or accuracy of the model. If this strategy still does not result in real-time performance, error messages are produced indicating to the operator that the distribution of models between the various processors may need to be changed.

### 3.8 Discussion

In this chapter a component-based framework, MulTiSIM, has been described, intended for use on a multi-processing network having relatively low-bandwidth communications. Its main characteristics are:
- It provides for a hierarchical decomposition of objects into constituent parts, with the constituent objects split between processors in an arbitrary manner.

- It is written entirely in Ada, with provision for multi-language working within an object, to encourage reuse of existing code.

- It provides generic communications between objects, both for objects to send information to others, and for objects to request information about others.

- The structure of the framework has been chosen to maximise the design independence of its various component parts: models, communications packages and the core framework itself. This is intended to minimise the propagation of consequential design changes from one component to another.

- The models are intended to be reusable in a variety of contexts. Their interactions are independent of class, distribution and constituent structure, and they can be formed into archives of cooperating models.

- The framework code is readily portable to different computing environments, encouraging the use of heterogeneous computer networks to optimise processor use.

The main areas of originality in the framework concern two of the three transparency principles applying to the interactions between models within the framework, transparency of distribution and structural transparency, as well as the use of Ada generic packages for the specification of inter-model communications and the load-smoothing aspects of the scheduler. The first principle, transparency over model type (or class) is regularly used within object-oriented systems to allow objects to possess components with varying characteristics, which have been derived from a single super-class template.

With hindsight, the distributional transparency principle has turned out to be an application of the good architectural principles advocated by Jackson & Boasson (1995) to the field of real-time simulation. It allows the design of models to be de-coupled to a large extent from considerations of the physical network on which they will run, without unduly compromising the real-time performance of the system.
The remaining principle, that of structural transparency, becomes important within MulTiSIM because of the way in which information is accessed by object constituent, rather than by entity, which can complicate the retrieval of information. Steps have been taken, both in the design of the framework and of the models which interact within it, to ensure that the operation of any one model does not become unduly dependent upon the component structure of any of the others.

The following chapter provides examples of the use of the framework by describing the main features of two real simulators which have been constructed within it. The framework is intended for medium-sized simulations, in which global broadcasting of information is practical. Analysis has shown this approach to have relatively poor scaling properties; alternatives with better scaling properties are discussed in Chapters 5 and 6.
Chapter Four
Chapter 4  Examples of Simulations Based on the MultiSIM Framework

This chapter provides two examples of distributed simulators which have been developed using the MultiSIM component-based framework, illustrating the approach to model design which the framework is intended to encourage. The first example is a small-scale driving simulation with relatively few components and a fixed set of information transfers between them. The second is an altogether more ambitious project to construct a full-scale helicopter mission simulator, in which a wide variety of models would be accommodated, and in which the use of information would be determined at run-time by various detection algorithms which were an integral part of each model.

4.1  The THRUST Super-Sonic Car Simulator

The driving simulator developed for Richard Noble's Thrust-SSC supersonic car project provides a simple example of the way in which a set of models for MultiSIM can be built up. The main purpose of the simulator was to familiarise the driver with the cockpit controls and the behaviour of the car's systems prior to initial low-speed testing.

The major components of the simulation software are shown in Figure 4-1. In this example, all communications were made through data stores, no discrete-event handlers being found necessary. The stores defined held driver's input values, engine controls, outputs from the engines and the position, attitude and speed of the car. The main vehicle dynamics model is fed by steering, brakes and parachute inputs from the driver and by thrust from each of the engines. The engine models were written as a class, with two instances for left and right engines taking throttle and starter inputs. The driver's inputs were derived partly from analog controls, and partly from a touch panel display. The out of cockpit view was generated on a Silicon Graphics RE2 workstation, using Iris Performer controlled through a MultiSIM harness.
Examples of MulTiSIM Simulations

Figure 4-1 Model Components Comprising the THRUST-SSC Simulator

A sub-set of the models was initially tested on a single machine, and finally integrated onto two machines, with inputs and instruments on one, dynamics models and out of cockpit view on the other. No changes to the models were needed during this integration, in line with MulTiSIM’s principles of distributional transparency.

The opportunity to do this work arose primarily through contact with Sqn Ldr Andy Green, the driver of the car, who was working on the JOUST simulator in DERA. The final version of the simulator was demonstrated on the THRUST stand at the Motor Show at Earls Court in 1995, where it was subjected to a ten day endurance test conducted by relays of small boys.

4.2 The HOVERS Helicopter Mission Simulator

The Helicopter Operational Visual Engagement Real-time Simulator (HOVERS) (Roden & Corbin, 1994) is a mission simulator covering helicopter nap-of-the-earth flight. In this context, ‘nap-of-the-earth’ refers to flight below 50ft, using terrain features such as woods, buildings and hills to conceal the helicopter from enemy forces. To present a realistic impression to the human aircrew, the simulator must have a three-dimensional terrain data-base containing a considerable
level of detail. The human aircrew are presented with views from their cockpit of this terrain, and any objects moving through it, in the visible and infra-red wavebands. These views are computer-generated from the information in the terrain data-base and in the MultSiM object data-base and data stores.

Careful use of the features of MultSiM has resulted in it being able to support HOVERS simulations involving over 400 separate objects (including component parts) comprising 50 independent entities on up to 25 processors running in real time using only a single 10Mbit/sec LAN for inter-processor communication. The communications load has been kept down to less than 50 Kbytes/sec for this simulator configuration, largely due to the information management provided by the framework. The simulator currently implemented provides the information needed by six aircrew, controlling three two-man helicopters, with thirteen 3D displays driven by texturing graphics workstations.

The models used are written in a variety of languages, Ada, Fortran and C, and vary in size from a few hundred lines of code (for a simple decoy flare) to over ten thousand (six degree-of-freedom non-linear helicopter dynamics model), a typical size being 2000 lines. Model update rates vary from 1 Hz, for some parts of the computer-generated ground forces, to 50-100 Hz used for helicopter and missile dynamics. Multiple update rates like this are accommodated within individual processors using the automatic load-smoothing provided by the scheduler.

No problems have been experienced from the potential inefficiencies which might have been expected due either to the use of Ada or to the use of object-based techniques within the framework. The optimising Ada compiler used has a run-time efficiency comparable with Fortran. The use of object-based techniques, such as dynamic binding, is largely confined to the controlling layer of the simulation, whereas the majority of the execution time is spend in the models.

The simulator needs an accurate representation of the physical processes involved in detecting helicopters and other vehicles at the various wavebands of interest (visible and IR), when they are partly concealed by terrain features. This representation must be consistent between the human aircrew, who are using their view of the outside world for detection, and simulated sensors used by the computer-controlled combatants (ground forces and other helicopters), which
Examples of MultiSIM Simulations

rely directly on the information available from the MultiSIM object data-base and data stores to model the detection process.

HOVERS is intended to be used in the future as one component in a synthetic environment, in which a number of suitably compatible simulators can be interconnected to investigate problems which cannot be solved by a single simulator. It has proved relatively straightforward to construct gateways between a MultiSIM network and other simulator networks, both within the local area network, connecting for example to a high-bandwidth motion platform and flight model running on a separate network, and over wide area networks using DIS protocols (Seidensticker, 1994).

4.2.1 Infra-Red Detection Modelling in HOVERS

Detection of an object in the infra-red will be used as an example of the use of data stores. This requires a knowledge of the effective temperature of the object in the IR waveband of interest. To be more precise, the temperature of each separate part of the object not currently obscured by the terrain should be known. This is because, at certain ranges, the ‘hot-spots’ which result from uneven temperature distributions will make the object visible, while the use of an average temperature for the whole object would falsely indicate that it is not. Temperatures change with time, depending on the activity that the object is engaged in.

The visibility calculation also requires knowledge of the effective temperature of the background, atmospheric attenuation, sensitivity of the sensor, and the degree to which the terrain is obscuring the object. The calculations are performed on the processors simulating the sensors, to ensure that they are carried out only on demand, and with minimal time delay. Constant information, such as the terrain data-base, is distributed before the start of a run.

To implement this scheme, a considerable amount of time-varying temperature information must be broadcast to all processors. Two characteristics of MultiSIM aid in this exchange and prevent it from causing saturation. The first is that each component part of an object is treated as another object in its own right, thus ensuring that it has its own entry in the data stores. This means that there is no inbuilt restriction on the number or complexity of components which can be represented within the object data-base. This inherent expandability is one of the advantages of taking an object-based approach, as distinct from the purely entity-based approach traditionally
Examples of MultiSIM Simulations

used in real-time simulators. It encourages the development of multi-scale simulations, where some behaviour aspects of particular importance to a trial can be modelled at a considerable level of detail, whereas others may be represented at a lower level of fidelity.

The other helpful feature is that a data-store entry is only rebroadcast when it has been updated by the associated dynamic model. The thermal models are designed only to update their data store entries when the temperature has changed by more than a threshold. Since temperatures change only slowly, and occasionally, the practical effect of this optimisation is to make the additional communications traffic needed to transmit temperature information negligible.

4.2.2 Representation of Articulated Objects in HOVERS

Another example of the use of MultiSIM data stores occurs in the representation of articulated objects. An articulated object is one with component parts which can move in certain ways relative to the main body, which is itself moving. A typical example would be a tank turret swivelling on the body of the tank. These components may, in addition, have articulated components of their own, to any level of detail needed by the simulation. Furthermore, in certain cases, the topology of the components can change dynamically, as when new components are created, or when an existing component, such as a missile, flies off and becomes an independent object. Finally, the component parts of an object must be capable of being arbitrarily distributed across the computing network.

The scheme adopted for the data-store which holds position and attitude in HOVERS allows any complexity of articulated component parts to be represented, and to be transmitted efficiently between processors. The scheme uses a variant record type which, for an independently moving platform, holds an absolute position and attitude in earth axes, and for an articulated component part, holds the offset in position and attitude of the component from its owner. In combination with the owner/parts information in the object data-base, this can be used to build up a tree of articulation information to any depth. The structure, as well as the contents, of this tree can be modified during a run, either by altering the variant record type, or by changing the owner/parts information dynamically.
An access procedure is provided which will combine the absolute and offset information through a rotational transform to give the absolute position and attitude of the component, when needed. This operates recursively to any depth to take account of multiple, nested, component offsets.

Only those components which actually change their articulation position need transmit offsets at run time. The majority of offset information is constant, and need be transmitted once only at the start of a run. This would apply, for example to the constant offset of a missile attached to a launcher. When the missile is launched, however, it replaces this offset with its dynamically changing absolute position, and can be perceived elsewhere in the simulation as a free-flying object. At the same time, this change can also be transparent to the other objects, since the original access procedure can still be used to obtain its position.

In addition, once a missile is flying free from its launcher, it may change its ownership within the object data-base, so that it is no longer a component part of the launcher. This ensures that if the launcher is destroyed, the missile will continue to exist. The alternative approach, taken by many simulators, is to create the missile object dynamically at the time of launch. However, this approach would not allow the missile to retain any information which its owner may have passed to it before launch.

4.2.3 Use of Protocols to Define Model Interactions in HOVERS

Models for the HOVERS simulator typically interact not through a single data store or event handler, but by using a combination of information from several data stores, possibly with some discrete events used as well when appropriate. When designing sets of interacting models for HOVERS, it became very difficult to appreciate the main features of the design due to the number of interactions taking place. A method was needed by which the model interactions could be expressed in summary form.

This was done by defining each set of data exchanges between two models to be a separate protocol, and documenting the models in terms of these protocols, rather than using the separate interactions making them up. A limited number of protocols (seventeen) has sufficed to describe all of the model interactions needed for the HOVERS simulator, there being considerable commonality between different models. Figure 4-2a shows the design of a set of models
Examples of MultiSIM Simulations

representing a group of tanks, described in terms of five of these protocols, which are defined in Figure 4-2b.

![Diagram of model interactions for a tank group described in terms of protocols]

**Figure 4-2a Model Interactions for a Tank Group Described in Terms of Protocols**

This diagram shows one possible way of combining these particular models. It is, however, possible to re-use them for different purposes, for example, the tank turret can be replaced with a missile launcher or an armoured personnel carrier without affecting the operation of the vehicle or group models; indeed, these different types can be mixed within the same group. The vehicle model can be used on its own, without a group controller, if necessary.
The use of protocols at the model design stage has considerable benefit in ensuring that different sets of models will interoperate successfully. It provides a means for checking that all the information needed is being generated, and that all events being generated will be handled correctly.
The ideas of structural transparency are compatible with the use of these protocols. The three items of information required by the Detection protocol, for example, could well be defined at different levels in a model hierarchy without affecting the operation of the protocol. Similarly, the Strike event generated within the strike protocol could be handled at a higher level than the target object which is apparently being attacked. This would indeed be the case if the object being attacked were, for example, a powerful emitter attached to a mobile platform. In this case the platform would need to handle the strike event, since damage would not be confined to the emitter component.

It is hoped to extend the use of protocols in future work. Where a protocol involves a number of discrete events, the ordering of these may become crucial to the successful operation of the protocol. It is hoped to tighten the definition of such protocols by incorporating state transition diagrams, covering the options available at each stage at both ends of the protocol. It has also been suggested that the protocol definitions could be rendered in packages of software, tightening the definitions still further to allow run-time policing of their correct operation, and removing a lot of the MultiSIM-specific details of the model interactions from the model code itself.

4.3 Conclusions

This chapter has demonstrated some the possible ways of making use of the facilities of the MultiSIM framework in constructing distributed simulations. The experience of using this component based architecture for real simulations has been very encouraging. It has proved possible to re-use existing components through a number of different simulator configurations designed for various trials, and to introduce new models with minimal modification to the old ones.

Vital in this, however, has been the stability of the designs for data-stores and event handlers underpinning the communications protocols. One potential flaw in the approach embodied within MultiSIM is this requirement to be able to define the nature of the communications between a set of models at a relatively early stage in model development. This theme will be returned to in chapter 8, where a mobile-code monitoring system requiring little or no agreement on protocol definitions will be described. The next chapter starts the consideration of mobile-agent approaches to simulation with the description of a system relying entirely on mobile entity models.
Chapter Five
Chapter 5  Pilot Study on Model Mobility within Simulations

This chapter describes a collaborative study undertaken by the author and Dr. Sapaty of the University of Surrey, with assistance from James Darling, to investigate whether mobile-program techniques, such as those embodied in the WAVE language, could be applied to distributed simulations. This description concentrates on the author's own contribution, together with as many details identified from the contributions of the others as are needed to place it in context. More information about the work can be found in Corbin & Sapaty (1995).

The main issue prompting this study was the relatively poor scaling performance of conventional distributed simulation schemes, like DIS and MultiSIM, which rely on global data exchange for their operation. As the number of entities in these simulations increases, and in the absence of any scheme for managing the distribution of information, the total processing load inevitably rises with the square of the system size, since each entity is responsible for performing its own relevance filtering on the information coming from all the others (Pullen & White, 1995). The idea of the work reported here was to design a scheme in which information management would occur automatically as an emergent property of the way in which the entities were distributed within the computing network and of the low-level interactions between them.

The first two sections describe the purpose and methodology for the study, with details of the prototypes developed being given in section 5.3. In section 5.4 a speed and scaleability performance model for the type of simulation developed is presented and compared with the equivalent model for more conventionally organised systems. Section 5.5 contains a summary and critique of the study.

5.1  Purpose of the Study

The main purpose of this study was to establish whether it would be possible to set up a simulation environment which would be indefinitely extensible, one which would allow models to communicate with those around them in virtual space wherever they may wander through the simulation arena. This scheme should not impose any increase in computing loads as the simulation scale increases, nor use any central resources which could become a bottleneck, e.g. for routing communications, but could operate over a heterogeneous wide area network in an essentially self-organising manner.
As an initial approach, it was decided to follow Dr. Sapaty's suggestion that a set of fully mobile models be constructed in the WAVE language, these being able to relocate themselves between the various nodes of the network to mirror the motions of the entities they represent within the virtual space being simulated. To do this the effective connectivity of the computing network must be at least sufficient to span the number of dimensions used to divide up the virtual simulation space. For example, in this work the simulation space was divided up as a two-dimensional map, so the network had to be configured as at least a 2D mesh in order to span it. If a volumetric partitioning had been attempted, then a cubic computing grid would have been necessary.

The mobile entity models should demonstrate the ability to exchange state information with other entities on their current node (i.e. entities which would be near to them in virtual space) and also to react to changes in a common environment within which the models were operating. Since WAVE is an interpreted language, and would not have been efficient enough to implement the equations of motion of the entities, it was decided to adopt a hybrid approach in which model dynamics would be written in C, and would operate under the control of an overall WAVE program which would be responsible for moving the entities between the various nodes of the network, as will be described in the next section.

The author's main area of contribution to this experiment lay in the devising of a mechanism for implementing the equations of motion of the models in a way which would allow them to operate in real-time, while still being under the control of a WAVE program as far as their network distribution was concerned, and designing the WAVE to C control and communication protocols. It also involved establishing the method used for inter-model communications, as well as that for allowing models access to information about the simulated physical environment in which they were all to operate. Finally, the author undertook the design and development of the main model classes, 'Aircraft' and 'Fighter', together with a set of utility facilities for operating the simulation.

5.2 Hybrid Distributed Simulation in WAVE

The WAVE language and interpreter can readily be used to implement simulations involving a number of object-based models, operating in continuous-time mode (though not necessarily in real time), distributed in an arbitrary way over a network. This distribution can be automatically
varied on a dynamic basis to mirror the spatial distribution of the entities. The model dynamics are prepared in a conventional language such as C, leading to a hybrid simulation in which the WAVE programming layer is used to control the instantiation, distribution and synchronisation of objects. In this way, each programming paradigm is used to best advantage; the inherent efficiency of the compiled language is needed for real-time execution of the dynamic equations, its encapsulation properties are utilised within each model to enforce code independence and permit re-use, while WAVE is used where maximum distributional flexibility is needed.

The key to this is the ability of the WAVE interpreter to invoke another Unix process. Within a WAVE program, this operation is controlled by the "?" operator, e.g. \texttt{Fn?process\_name} will invoke a Unix process, passing it a string in the variable "Fn" containing command-line parameters. The current WAVE program thread then suspends its operation until this new process has finished, and receives any output from it written back into "Fn" on return. Other threads of the WAVE program continue to operate concurrently.

![Diagram](image)

\textbf{Figure 5-1} Subdivision of the Simulation Arena into Discrete Spaces

The simulation scheme described below uses a thread of the WAVE program to represent the localisation of the entity being modelled. This thread is free to travel over a graph whose nodes each represent a region of the simulation space (Figure 5-1). When the WAVE program enters a fresh node of the graph, it launches a new Unix process able to solve the dynamic equations of
motion of the entity, which can then be used to model its behaviour within that node’s region of space with full accuracy.

The state information needed to initialise this dynamic model is carried with the moving wave in frontal variables, and given to the Unix process as a command string when it is launched. Typically, a model will run until it reaches the boundaries of the region represented by that node of the graph. At the termination of this process, the final values of the states are returned to the WAVE program, and can be used to launch the model in another spatial node whose identity can be determined from the final state values.

A case of particular interest occurs when more than one model is present in the region represented by a node, and these models are, moreover, required to interact in some way. Each model is represented by an independent Unix process, but they are able to exchange the data required for any interactions via a shared memory region, reserved for that node (see Figure 5-2). In effect, this shared region becomes a temporary object data-base containing an entry for each of the models on the node. The information in the entry is updated regularly by the Unix process calculating its equations. The other objects are able to inspect the data in each entry and use it to, for example, implement a pursuit/avoidance interaction.

This object data-base can also be used by other wave programs to inspect the state of the objects being modelled - an additional Unix process is invoked to return the information. In this way it is possible to implement a supervisory layer of wave programs, able to analyse the overall status of the simulation by combining information gleaned from a number of geographically separated nodes, or to track individual entities, as in the “roaming cockpit” concept suggested by Sapaty (et. al. 1995a p.204). The monitoring functions which can potentially be implemented are not pre-determined at the start of a simulation, but may be specified by any of the users, who may dispatch a WAVE program containing the code required to perform the analysis to all relevant nodes. This illustrates one use for the flexibility inherent in the WAVE paradigm. Figure 5-3 shows in schematic form the major processes happening within the WAVE programs performing the model control and analysis functions.
Objects may also be controlled through the shared-memory object data-base. The most obvious mechanism is to force the early termination of a modelling process by setting a "kill" flag in the data-base. This can be done either by the supervisory wave layer, or by another model, for example, when its pursuit is successful.
a) parameters defining the model and its internal behaviour, e.g. its name, its route, or its target.

b) current values of states, e.g. position, heading, speed, which also have to be given values on initialisation.

c) mapping information, indicating to the C model layer which shared object data-base should be mapped to next.

d) termination conditions, indicating to the WAVE layer the reason for return from the C layer, and determining the subsequent behaviour of the model.

These variables may be interpreted as either strings or fixed-point numbers, depending on context, e.g.:

\[
\begin{align*}
\text{Fn} &= \text{aircraft}_x, & \text{name of model}, \\
\text{Fx} &= 345, \, \text{Fy} = -1024 & \text{position coordinates in metres}.
\end{align*}
\]

The WAVE vector construct can be employed to pass the values of records and arrays, or to initialise the contents of lists and stacks. For example, the route for the aircraft model described below is passed as a series of waypoint coordinates within a single vector “Fw”:

\[
\text{Fw} = 30;29;-500;-500;-1000;647.
\]

This is interpreted by the C modelling layer as the contents of an array of records, each record containing a pair of numbers referring to a single waypoint. The variables may be given in any order, and need not always be present. Their absence can either be taken as a default condition (as in Aircraft model, see below), or can be used to trigger alternate models of behaviour (as in the Inspect utility program).

These constructs were intended to provide a flexible mechanism for intercommunication between the different layers of the multi-lingual modelling environment. The information transferred need not be confined to the straightforward variables used here, but could include, for example, file references or URLs for data-base files containing geographical or tactical information sets.
Such controlling and communications mechanisms are fully consistent with the multi-user aspects of the WAVE paradigm, since models created by different users can navigate over a common spatial graph, but can create independent Unix processes as dynamics models. When required to interact at a detailed level with the models belonging to other users, each of these has full access to the shared local object data-bases.

5.3 Description of the Mobile Models

The initial set of modelling software developed by the author comprised two varieties of active models, aircraft and fighter, and a number of utilities useful in the management of the system.

**Aircraft:** This model contains the speed and heading dynamics of an aircraft. At launch it is given initial values of its states and a route to follow, comprising a set of waypoints, and must steer itself between them. In general, this route will cross several boundaries between nodal regions (see Figure 5-1). When a model reaches a boundary, all internal information held by the model is returned to the WAVE layer in a string which is passed on to the next dynamics model in the new node. Thus on entry, the control string passed to the aircraft model might be, for example:

\[
N=s1.Fn=air1.Fy=2.Fx=7.Fh=45.Fs=57.Ft=0.Fw=23;45;200;25;300;300
\]

specifying the name of the node on which it is to operate, its own name, position, heading, speed, time reached and waypoints. The aircraft model will open the shared area relevant to that node, and inspect the positions of the boundaries of the geographical area covered by the node. When it passes over one of these boundaries during execution of its equations of motion, it will stop executing and return a final string such as this:

\[
Nb=3.Fn=air1.Fy=220.Fx=75.Fh=50.Fs=58.Ft=40.Fw=23;45;200;25;300;300
\]

in which the first parameter is the number of the boundary which has been passed, which is used by the WAVE layer to deduce the identity of the new node from the labelling of the links on the spatial graph, and the remainder contain the updated information needed to restart the model on the new node, once the controlling WAVE program has moved across the link logically connecting the two nodes (Fp is the number of the next waypoint in the list).
While it is operating, each aircraft periodically writes its position into the shared area, which acts as an object data-base. This information can then be used by other models, such as fighter below, to interact with the aircraft.

**Fighter:** The second type of model is similar to aircraft, but has the additional property of being able to pursue one of the aircraft and destroy it. The identity of this enemy is passed to the fighter as an additional variable in its command string (\( Fe = \text{name_of_enemy_aircraft} \)). It follows a track defined by waypoints, just as the aircraft does, until it enters a region of space where its opponent is. Once it has detected its opponent by finding that its name has entered the object data-base, it enters pursuit mode and tries to approach the opponent close enough to destroy it, using position and heading information left by the opponent in the object data-base. Should the opponent pass through a boundary to another region, the fighter will attempt to do the same to continue the pursuit, otherwise it reverts to its original waypoints.

Other programs were also developed by the author to establish and manage the simulation:

**Start:** This sets up an empty object data-base for one of the regions, and records the positions of the boundaries as a header to its object data-base. The boundary positions are transmitted as triples within a WAVE vector, i.e.:

\[
\text{Nb}=\text{north1;east1;orientation1;north2;east2;orientation2; ...}
\]

**Inspect** returns a list of the names of players currently in a region, or the position and heading of one of them, if given its name as a parameter. **Alter** allows the rate of integration of the players to be adjusted while **Kill** forces the immediate destruction of one of the players and **End** cleans up and removes a shared object data-base.

This system was operated over a small network of processors on the local area network within Surrey University, and also over wide area network connections using the Internet.

**Environmental Modelling:** A further development of this system was the introduction of a common environmental model which changed the behaviour of all of the entities in responding to it. The environment was dynamic and took the form of one or more clouds, moving through the space occupied by the entities, which they had to avoid. The clouds were represented by irregular polygons, and could transition between the processors making up the network in the course of their motion.
The intention behind this work was to demonstrate the potential of WAVE for constructing models of the environment which would operate in a distributed fashion, with different sections of the model on different processors. This would permit the development of models which could be extended to cover indefinitely large areas, while retaining the possibility of dynamic modification to individual features by ensuring that they occur in only one place in the model. This extension of the study into modelling of the common environment led on eventually to research into dynamic terrain modelling, conducted by James Darling.

The author's contribution to this was to implement an extension to the shared object data-base within each processor to accommodate sets of coordinates defining the shape of the sections of cloud(s) currently occupying that processor's area of space, together with the WAVE/C hybrid facility which updated the entries in this data-base as the cloud progressively entered the region, moved through it and finally exited through the opposite boundary. A feature was also introduced into the Aircraft model by the author for detecting the presence of one or more clouds blocking the route to the entity's next waypoint, and calculating the minimum deviation from track needed to carry out a manoeuvre to avoid them (Sapaty, Borst, Corbin & Darling, 1995b).

5.4 Performance Models for Large-Scale Simulations

In this section are presented two theoretical performance models for contrasting architectures of distributed simulations developed by the author. The intention of this is to illustrate the theoretical differences between a conventional broadcasting scheme and the type of dynamic distributed scheme described above. In addition it is intended to demonstrate that these differences are significant when used with reasonably practicable parameter values.

5.4.1 Model for Systems using Global Data Broadcasting

The first performance model is applicable to conventional systems which employ global broadcasting of information by each simulator, and in its broad features would be applicable to basic DIS systems and MultiSIM, as well as to a number of other multiprocessor simulators. The model includes four processing speed parameters:

a) the time taken to run each entity model, in the absence of additional information from other entities, \( t_e \), and including the time taken to broadcast its updated state information,
b) the additional time taken by each entity model to assess the information from the other entities in the simulation in order to interact with them, $t_i$.

c) the overhead time involved in running each computing node, in the absence of entities, $t_n$ and,

d) the additional time taken by each computer node to read in the information from entity models broadcast by other nodes, $t_r$.

In addition, there is a fifth parameter giving the maximum capacity of the broadcast data-bus. This model has been used to estimate the maximum update rate achievable by the simulation for various numbers of processing nodes, $n_p$, and various total numbers of entity models, $n_m$.

The processor cycle time is given by:

\[
\text{cycle time} = \text{node time} + \text{entity execution time} + \text{data reception time},
\]

where:

- node time \( = t_n \),
- entity execution time \( = (\text{no. of entities on node}) \cdot (\text{total time per entity}) \)
  \( = n_m/n_p \cdot (t_e + t_i \cdot (\text{total number of other entities})) \)
  \( = n_m/n_p \cdot (t_e + t_i \cdot (n_m - 1)) \)
- data reception time \( = t_r \cdot (\text{number of entities on other nodes}) \)
  \( = t_r \cdot (n_m - n_m/n_p) \)

Thus, the processor update rate in Hz is the reciprocal of the cycle time and is given by:

\[
( t_n + (n_m/n_p) \cdot (t_e + t_i \cdot (n_m - 1)) + t_r \cdot n_m \cdot (1 - 1/n_p) )^{-1}
\]

Figure 5-4 and the tables in Appendix C show the major features of the results when data-bus saturation is not considered. Parameter values used were $t_n=2$ msec, $t_e=1$, $t_r=0.05$, $t_i=0.025$, though the overall form of the results is not at all sensitive to these values. The most notable feature is the way in which the performance diminishes rapidly as the total number of entities involved in the broadcasting of information increases, dominated by the time needed by each entity to assess the data from the others. The graph shows that it is not possible to win back
much performance by increasing the number of processing nodes, since the load on each node is
heavily dependent on the total simulation size - the parameters $t_i$ and $t_r$ multiplied by $n_m$
eventually dominate the performance in systems of this sort.

Figure 5-4 Update Rate for a Distributed Simulation with Global Broadcasting

The effects of data bus saturation will now be considered in addition. The communications
network cycle time is given by:

$$(\text{network availability factor}) \cdot t_s \cdot (\text{total number of broadcasts})$$

where the network availability factor has been introduced to account for the characteristics of
some networks in which the full theoretical bandwidth is not available. For example, Ethernet
does not pre-allocate bandwidth, but allows processors to broadcast at will, limited only by lost
messages when two such broadcasts coincide and clash. In this case it is difficult in practice to
make use of more than one quarter of the theoretical bandwidth of $10^7$ bits/sec. Thus, for an
ethernet system, transmitting HOVERS position and attitude packets of 320 bits, the update rate
is given by:

$$(4.0 \cdot 320 / 10^7 \cdot n_m)^{-1} = 0.78 \cdot 10^4 / n_m$$
Thus, this bus system could accommodate 300 entities operating at 25 Hz. The carpet plot of Figure 5-5 allows for data bus saturation effects by plotting the minimum update rate given by the processor and bus formulae.

![Carpet plot diagram](image)

**Figure 5-5 Update Rate for a Distributed Simulation with Broadcast Bus Saturation**

Note that the above does not take account of the increase in bandwidth which can be made available by the use of the dead reckoning techniques defined as part of the DIS standard. In favourable cases, such as a slowly manoeuvring ground vehicle, this allows the use of update rates as low as 0.5 Hz for communications. The advantage is not so apparent for rapidly manoeuvring vehicles, however, where dead reckoning can introduce unacceptable artefacts into the perceived motion (Crush, Corbin et al, 1996). The effect of dead reckoning on processor update rate is entirely negative, since the receiving processor needs to take additional cycle time to reconstruct the high update rate motion from the low update rate data transmissions.

As a point of reference from a practical system, the HOVERS simulator, using the MultiSIM framework from Chapter 3, operates in the region of 64-128 entities, 8-16 processors and 20-30 Hz, which corresponds reasonably well with the 4th and 5th curves in Figure 5-5.

### 5.4.2 Model for Systems using Geographical Mapping

In contrast to this, the second performance model applies to geographically mapped architectures, such as the one described in this chapter, in which no information is broadcast
globally. This model assumes that the majority of the information needed by an entity model can be found on its local processing node, with an area of overlap extending into adjacent nodes. This overlap information is received over a network consisting of independent data-links between the nodes. An additional parameter is introduced to describe the overlap:

\[ n_i \]  
number of entities whose data is of interest to each other entity.

Again, the processor cycle time is given by:

\[
\text{cycle time} = \text{node time} + \text{entity execution time} + \text{data reception time},
\]

but some of the terms have altered.

Each entity on a node will need to inspect data from all the others on the node, plus those from the overlap region. It is possible to estimate the total number of entities of interest to the node if we also assume that they are approximately evenly distributed in virtual space. A circle of radius proportional to \( \sqrt{n_i} \) would contain the entities of interest to a single other entity. A circle of radius proportional to \( \sqrt{(n_m/n_p-1)} \) would contain the other entities local to one node, and so a circle of radius proportional to the sum of these two, i.e.

\[
(\sqrt{n_i} + \sqrt{(n_m/n_p-1)})
\]

would contain all entities of interest to the node. Hence the model execution time becomes:

\[
(\frac{n_m}{n_p}) \cdot (t_e + t_i \cdot (\sqrt{n_i} + \sqrt{(n_m/n_p-1)})^2)
\]

If we assume that the transmission architecture is efficient enough to eliminate unwanted data transmissions at source, then the data reception time depends on the total number of entities outside this node which are of interest to those entities which are on this node. This number is much less than \( n_i \cdot \) (number of entities on node), since many of the external entities are of interest to several of the local entities, being in the same geographical area. In fact:

\[
(\sqrt{n_i} + \sqrt{(n_m/n_p-1)})^2 - (n_m/n_p-1) = (n_i + 2 \cdot \sqrt{n_i} \cdot (\sqrt{(n_m/n_p-1)}))
\]

would be external to it. Hence the overall processor update rate becomes:

\[
(t_n + (\frac{n_m}{n_p}) \cdot (t_e + t_i \cdot (\sqrt{n_i} + \sqrt{(n_m/n_p-1)})^2) + t_r \cdot (n_i + 2 \cdot \sqrt{n_i} \cdot (\sqrt{(n_m/n_p-1)})))^{-1}
\]
Using the value \( n_i = 30 \) and with the same values for the time parameters, gives the performance curves shown in Figure 5-6 and Appendix C. It has not been thought necessary to allow for data bus saturation since in systems of this sort, where transmission is essentially local, advantage should be taken of this fact to provide a scaleable system of independent local busses.

![Figure 5-6 Update Rate for a Geographically Sorted Distributed Simulation](image)

The main change is that the performance obtainable is effectively invariant with the scale of the simulation - as the geographical area is expanded and more entity models are added, the same level of performance can be maintained by increasing the number of computing nodes. No data-bus saturation effects need occur, and the load on each processor remains sensibly constant as the system scales up, once the number of entities exceeds \( n_i \).

It is interesting to note that if, instead of plotting the absolute performance of the multi-processor system in Hz, a plot is made of the performance relative to the equivalent single processor system, then a very different impression of both systems would have been given. The curves in Figure 5-7 and 5-8 have been plotted on this basis. The first exhibits an apparently impressive linear speed-up as the numbers of processors are increased. This completely masks the fact that the absolute performance of the single-processor system in Figure 5-7 drops well below real time on large simulations, and that the linear speed-up is nowhere near sufficient to restore the system to real time operation.
The speed-up for the geographically sorted system is in fact super-linear, this apparent impossibility reflecting the fact that the total amount of processing to be performed diminishes as the system is subdivided into smaller parts and the effects of the geographical sorting in eliminating nugatory interactions become more significant.

5.5 Discussion and Critique

This is believed to be the first time that a system has been demonstrated in which sets of dynamics models were allowed to migrate over a network as an integral part of their operation. The work demonstrated that this could be achieved within a hybrid WAVE/C programming environment, albeit with relatively low order linear dynamics models. The beginnings of a distributed environmental modelling scheme were also demonstrated.

The significance of this study perhaps lies perhaps not so much in the relatively limited objectives which it was intended to achieve, but in the questions which it raises when considering the application of similar techniques to more complex practical simulations.
Figure 5-8 Speedup for a Geographically Sorted Distributed Simulation

The first issue to arise from this work is the need to accommodate more realistic interactions between models. The scheme above allowed access to information about entities within the area covered by the current node, but not to those outside of it. This results in entirely artificial limits on the operation of any sensor models included. Any practical scheme should give access to entity state information out to any range required by the operation of a sensor, preferably using mechanisms which are entirely transparent to the designer of sensor models.

The second issue concerns provision for human operators, particularly pilots of air vehicles who have arguably the most demanding roles in any simulation. Allowance will need to be made for operators to control their entity models from fixed stations, with constraints on the communications performance determined by the nature of the piloting task. If a pilot is to retain control of a vehicle with complex dynamics in a high-gain situation (e.g. landing or formation flying) then there is a limit of less than 100msec on the tolerable time-latency for motion feedback of his own vehicle, and a similar limit for motion feed-backs from the other vehicles with which it is interacting closely (IEEE, 1994c). This in turn largely determines the nature of the communications topology, since the only way in which these constraints can be met within current networking techniques is by direct communication from entity model to operator's station. The "roaming cockpit" concept described by Sapaty et. al. (1995a, p.204) relies on relaying of information through agents, and could not currently hope to operate with time latencies as short as those required for piloting.
Another issue concerns the arrangements for load balancing between the various processors on the network. The scheme above would result in considerable variations in processor loading over time as the various entity models concentrate their positions in different parts of the simulation arena. Any practical scheme should attempt to even out or at least limit the extent of these imbalances.

The overall robustness of the final scheme must also be taken into consideration. Mobile agents must be considered to be more fragile objects than fixed simulators, being more vulnerable to temporary malfunctions of the network. The loss of one or more mobile agents could seriously affect the outcome of a simulation. Fortunately in a mobile agent system there is considerable scope for increasing robustness by appropriate use of cross-monitoring between agents operating in different parts of the network. However, this needs to be considered at the outset, rather than introduced at a late stage in any design. A design for robustness will be discussed in Chapter 7.

A componency scheme, similar in concept to the one developed for MulTiSIM (chapter 3), will be needed to show clearly which are the permanent interfaces and facilities of the system, and which parts can be exchanged for various different simulation purposes. This is likely to be more complex than in the static-code case, since one of the most important modes of operation is for mobile agents to replicate themselves and spread over a system, communicating via purely private protocols, and evolving into what is effectively a distributed component, whereas in the MulTiSIM case components are localised and all communication takes place via the framework, through pre-defined protocols.

Finally, it should be emphasised that in the pilot study only the identity and current state of these entities were carried by the WAVE mobile code layer. The code describing the behaviour and dynamics of an entity (its model) was contained in a conventional executable file, produced for the particular computers in use, and down-loaded from a local file server when needed. Thus, although the dynamics models can be said to have been mobile, they were only mobile within the confines of a local network consisting of a homogeneous set of platforms. Some different operating principles will be needed before these limitations can be overcome, a subject which we will return to in Chapter 8, when the use of the Java language is discussed.
5.6 Summary

This chapter has described a pilot study involving the first use of mobile simulation models in a distributed simulation which, though it made use of a very limited sensing model, could nonetheless demonstrate a variety of behaviours. A performance and scaleability model was derived and used to verify that the system had no bottlenecks which could prevent it being scaled up indefinitely.

The main outcome of this study was to expose a range of issues which need to be tackled for any system of mobile models to be fully practical, in particular the need for more realistic multi-range models of sensing and detection processes, the need to make provision for non-mobile elements, particularly human operators, the need to consider the load balancing, robustness and stability properties of the system and the need to address the portability and component issues of the design.

The next stage of this work was to attempt to tackle the majority of these issues in the course of the design of a new architecture, in which the operation of the simulation was to be controlled by interactions between mobile agents which fulfil various distinct generic roles, to be described in the next chapter.
Chapter Six
Chapter 6  Generic Roles for Mobile Agents in Large-Scale Simulation

6.1  Background

The mobile models experiment described in the previous chapter illustrated the possibility of using mobile code to aid in the development of large-scale simulations. However, this work suffered from several limitations from the point of view of the designer of a practical simulation support system:

- it lacked any provision for modelling the operation of medium and long-range sensors;
- it assumes that all models are small enough to be rendered mobile through the use of an agent programming system, and that any model will operate successfully on any network node;
- it does not make provision for models to be operated by human crews, who must be presented with a low-latency interface, fixed in space.

In this chapter an attempt to overcome these limitations will be presented. It will describe a hypothetical simulator architecture based on the use of novel mobile agents to control the configuration and operation of the models making up simulations. The main objectives of this architecture are:

- to respect all of the constraints within which the simulations must operate. These may include use with particular pieces of hardware or with data-bases of limited availability, and the need to include human operators, who demand real-time operation from static sites;
- to ensure that the simulations will be indefinitely extensible, and not suffer from any restriction on overall size due to excessive demands on either processor performance or communications bandwidth;
- to ensure as far as possible that the simulations are flexible and reconfigurable by taking advantage of the highly dynamic characteristics of systems based on mobile agents.

In section 6.2 the proposed architecture will be described in terms of a set of generic roles which, in any real implementation, would be performed by specific types of mobile agent. It is on the
interactions between the agents fulfilling these different roles that the operation of the system depends. The set includes agents which perform interest management (controlling the exchange of information between simulation entities), management of computing resources (allocating entities to processors initially, and re-locating them dynamically where possible), dynamic modelling of environmental phenomena within the simulation arena, information extraction and collation, and mediation of specialised interactions between remotely situated simulation entities.

Section 6.3 provides an illustration of one way in which this architecture could be applied, in this case to the complex problem of multi-fidelity simulation. In an aggregated simulation, such as those used in ALSP federations (Weatherly et al, 1991), complete military units comprising many entities are represented by a single model. When such models are used within the same synthetic environment as entity-level models it is sometimes necessary to deaggregate them to entity level for interaction with the other entities and subsequently re-aggregate them back to their original form.

Section 6.4 summarises this part of the work, and introduces the prototyping work described in the following two chapters, in which the feasibility of certain of the generic roles is explored in more depth.

Little work on the application of mobile agents to the organisation of distributed simulations seems to have been reported in the literature. Work on swarm intelligence in the management of communications networks has already been mentioned, together with the work of Stone et. al. (1996) and Lubes & Valentino (1996) in Section 2.3.3.

6.2 Description of the Generic Roles

In this section a number of potential applications of agent-based techniques in the area of large-scale simulation will be discussed. These can be put into three broad categories:

- enhancements to modelling;
- simulation organisation;
- situational analysis.
The distinctions between these categories will be seen later to be somewhat blurred, but they form a useful initial framework for the discussion.

Enhancements to modelling may include a variety of topics, for example:

- provision of remote operators’ interfaces, specialised to particular models;
- reduction of time latency effects through the use of specialised prediction algorithms;
- modelling of various aspects of the external environment within which models exist and through which they may interact;
- control of aggregation/de-aggregation for aggregated models, used to represent the behaviour of collections of objects in simulations with a mixture of fidelities.

Simulation organisation can be regarded as including:

- control of the flow of information between those model entities which need to interact directly;
- assignment of computing resources to model entities (localisation of executable code);
- re-localisation of entities during the course of a simulation, to reduce communications loadings or for other reasons.

Finally, distributed systems for monitoring and analysis of data will be needed in large-scale simulations, since there will no longer be sufficient global data to allow these functions to be performed at any single site. Information on the “state of play” within the simulation is required both for immediate control, and for subsequent analysis. Similar distributed mechanisms may also be used within command and communications models to simulate the process of obtaining information from widespread networks of imperfect sensors and presenting it to high-level commanders.

In the conceptual system design, these various functions are obtained as an emergent property which arises from the interactions of several different types of agent. Each of these agents has a fundamental (or generic) role to perform, which can be modified in accordance with the needs of the particular models being supported, so that the role is performed in a specific way which may be different for each agent.
6.2.1 Mediation of Entity Interactions

The first area to be considered under the heading of modelling enhancements is the use of mobile agents in setting up and mediating the specific interactions which take place between closely-coupled entities within the simulation. The agents which perform this function may be termed "Interaction Agents".

Two general areas have been identified where the application of interaction agents may be of benefit:

a) the interactions of an entity model with a human operator, allowing the state of the model to be monitored and controlled, an Operator Interaction Agent, and

b) the interactions of two entities with closely-coupled dynamic behaviour, an Entity Interaction Agent.

Operator Interaction Agent: this type of agent is intended to relieve the limitations of distributed simulation frameworks in the area of monitoring and control of individual models. In general, it is difficult to establish the internal state of an entity model executing in a remote environment, except in rather general terms. For example, the DIS Version 2.0.4 standard contains a Data Query Protocol Data Unit (PDU) through which a number of items of information can be requested from a model. These items are restricted to those taken from a list of some 200 items defined in an enumeration (IEEE, 1994b, p. 287-294). This list has been designed to include a few attributes from all areas of modelling likely to be encountered, but in general only a small proportion of any individual model's internal states will correspond to items on the list, and so only part of the model's behaviour can be monitored.

The MulTiSIM framework (Chapter 3) has similar problems. In this case, the monitorable data can be any item which can be placed in a data store, or transmitted as an event via an event handler. Here, there is no inbuilt limitation on what can be sent, as there is in the DIS standard, since the simulation designer is free to define whatever data-store contents are required. However, in a practical system, it would rarely be possible or desirable to include all model states in a set of data-stores. Quite apart from the size, the set of stores would need re-defining whenever a new model was introduced, or an old one modified.

The operator's real requirement is for a system which could:
a) display all or a sub-set of a model's states and parameters in a readily assimilable form;

b) allow control to be exercised over both the initial and dynamic values of chosen states and parameters;

c) be capable of operating over a wide area network of limited bandwidth;

d) be capable of doing this for a wide variety of models, developed and maintained by other organisations, without prior knowledge of their form;

e) not be restricted to use within any particular modelling framework.

The ideal form of solution would appear to be a portable Graphical User Interface (GUI), which would contain displays and controls specialised to a particular model, and would be capable of being exported over the wide-area network to function on a remote general-purpose operator's station. Conventional distributed-graphics environments, such as X-windows, cannot be used over wide-area networks due to bandwidth limitations, and so some form of mobile-agent technique requiring a bare minimum of communication between the model and its agent would seem most appropriate. This agent would be produced and maintained by the organisation responsible for the model, thus satisfying requirement (d). This solution was suggested to the author by the capabilities built in to the Java language, and is here applied for the first time to distributed simulation.

Since it is concerned solely with a single type of entity, such an interaction agent can have a highly specialised interface to its model, allowing access to internal states and parameters which are not normally visible to other parts of the simulation, as illustrated schematically in Figure 6-1. An example of a prototype Operator Interaction Agent will be described in Chapter 8.

Figure 6-1  Schematic of an Operator Interaction Agent in use
**Entity Interaction Agent:** this type of agent would be used to assist in the interaction between closely-coupled entities. It could, for example, be used to perform specialised prediction calculations, based on intimate knowledge of its entity's current and likely future behaviour, in order to reduce the adverse effects which time-latencies in wide area communications networks have on simulation fidelity. The most practicable way of making use of this would be to call on such entity interaction agents only when two entities have particularly closely-coupled interactions, leaving the majority of interactions to be dealt with by simpler predictions, such as those defined within the DIS standard (IEEE, 1994a). The need for improved responsiveness in close-coupled flying tasks was one of the main recommendations coming out of Crush et. al. (1996), in which two fixed-wing combat simulators were linked over a wide-area network in order to assess fidelity for piloting purposes as a function of bandwidth and prediction parameters.

Figure 6-2 shows a schematic of the way in which entity interaction agents would be used, with the two entities exchanging their respective interaction agents once it becomes clear that their behaviour has become closely-coupled, e.g. when they enter into formation flying or a combat engagement. Each agent can then produce low-latency local reactions to the behaviour of the other, based on information obtained from a communications link with its originating entity. Since each agent would be specialised to represent the behaviour of just one model, and would probably possess low-order dynamic equations representing it, the nature of the information passing over this link need not be confined to any particular standard, but could include any internal states of the entity useful in making predictions of its behaviour (e.g. pilot inputs or moding information).

![Figure 6-2 Schematic of the Use of Entity Interaction Agents](image-url)
The two varieties of interaction agent described above are potentially capable of being used within any distributed simulation framework, since they are concerned only with private communications between individual pairs of models or with an operator's station. The agents in the rest of section 6.2 are concerned more with the overall organisation of the simulation, and would take over some of the roles of any simulation framework.

6.2.2 Communications Management

In any large-scale simulation, the management of communications is crucial to success. The straightforward global broadcasting strategy originally adopted by DIS (Seidensticker, 1994) has already been shown to be inadequate once the total number of entities exceeds the limits determined either by the total capacity of the communications medium or by the time taken by each processing node to absorb the information from all remote entities (Bassiouni et. al., 1995).

Any scaleable implementation of an infrastructure for communications management should avoid the need for global assessment of the possibility of interaction between entities, which increases the computing demands on each processing node as the total number of nodes increases, resulting in a square law for the total processing as a function of simulation scale. The majority of these interactions are physically impossible due to the nature of the sensing mechanisms involved, but must still be eliminated by explicit calculations in conventional schemes like DIS. The ideal system would so organise its use of information that the majority of these interactions need never be considered at all.

Any successful solution to the problem of communications management should attempt to satisfy a number of requirements:

1) The solution should as far as possible be generic, independent of the precise nature of the models or the nature of the data interchange between them.

2) The solution should be scaleable to any size of simulation. This implies that a fully distributed architecture be adopted, with no need for any central control which could act as a bottleneck, and that the loads on the individual elements of the distributed system should not increase with the scale of the simulation.

3) The loads on individual computing elements should also not be unduly affected by changes in the distribution of models in virtual space.
4) To be efficient, the solution should aim for maximum selectivity in communications, not transmitting anything which is unnecessary to the receiving entity. This implies a detailed knowledge of the nature of the sensors being simulated by the receiving entity, as well as a knowledge of the environment in which they are operating. The needs of the receiver for particular update rates and data freshness should also be taken into account.

5) The solution should work with any reasonable topology of network, and not be constrained to, for example, rectangular grids.

6) The solution should exhibit a considerable degree of fault tolerance. Single point failures in either transmission system or processing should produce no more than localised effects on certain models, from which a full recovery can be rapidly achieved.

The most promising approach to the design of such a system is based on the use of a geographical mapping between areas of the battle arena and computing resources within the simulation (Figure 6-3). The justification for this is that one of the principal factors determining whether or not two entities will interact is their relative positions within the simulated battle area. Entities which are situated close to each other are very likely to interact via any available sensing mechanisms (visible, infra-red, radar, sound etc.), while entities which are widely separated are much less likely to do so. The boundary between these two regions is determined by the individual characteristics of each sensing mechanism and by environmental factors such as the nature of the terrain and the clarity of the atmosphere (as noted in requirement 4).

The basis for this scheme is a set of processing nodes (or "agent meeting places", White, 1994), each of which corresponds to a section of the battle area. Over this network of nodes, agents representing each entity navigate so as to maintain a position in the network corresponding to the position of each agent's entity in the battle. We will refer to these as "Registration Agents", since their main purpose is to register the existence of their entity in a particular area (Figure 6-4).
The registration agent must maintain a point-to-point communications link with the simulator where its model is operating, through which it receives updates of position and status, and sends back controlling information to the entity. A registration agent may represent a complete group of cooperating entities (see section 6.3).

The most important function which the registration agent performs is to construct a list of those other entities in the simulation which currently need to interact with its own, and are therefore interested in receiving updates of its position and status. To build up this interest list, the
registration agent needs to perform a *discovery* operation, inspecting all entities whose sensors are potentially capable of detecting its own entity.

In current DIS systems, where entities are randomly distributed over the computing network, such discovery operations may involve a global search, which does not scale well. In the agent system being described here, advantage can be taken of the geographical sorting performed by the registration agents to restrict the area of search to the current processing node and those in its neighbourhood, out to a limit determined by the range of the sensing process being simulated.

In this scheme, discovery is performed as a cooperative process involving the registration agents for both the detecting entity and the target entity. The detector's registration agent sends out an *"Interest Agent"*, which has details of the characteristics of the sensor used to perform the detection, while the target's registration agent sends out an *"Attention Agent"*, which has details of the signature of its entity (i.e. how easy it is to detect). Discovery takes place when the interest agent meets the attention agent, and estimates that detection would be possible under the prevailing circumstances.

This discovery can then be reported back to the registration agents, which accumulate two lists, one of targets seen and the other of detecting sensors. These lists are then used by their respective simulators to set up direct communications paths between them, resulting in an overall low time latency compared with any information which could be obtained by using the agent as a relay. This is essential for real-time piloted operation. These communications paths may be individual point-to-point links, or multi-cast groups (Pullen & White, 1995) in which a broadcast mechanism is used to a sub-set of entities, resulting in a sharing of network traffic in cases where bandwidth is restricted.

This is an example of an interest management scheme, which should be distinguished from relevance filtering schemes (Bassiouni et. al., 1995), in which communications are selectively eliminated from a comprehensive set of broadcasts to reduce the total traffic volume. Relevance filtering schemes suffer from problems of efficiency, since the extent to which they can be applied depends on the fortuitous positioning of entities within the communications network, and problems of accuracy, due to the danger that the very information needed to make filtering decisions at one place in the network may have been removed by a previous filter at another part of the network.
In this scheme, the functions of the registration agents are largely generic, whereas the interest agents and attention agents are specialised to the nature of the sensor and target. The area searched by an interest agent will depend on the sensitivity and directional nature of its associated sensor. Similarly, the area covered by an attention agent will increase as the signature of its associated target entity increases, and may also have corresponding directional properties. For many short-range interactions, the functions of attention agent and registration agent may be combined to simplify the system (see Figures 6-5 and 6-6).

Chapter 7 describes the implementation of a prototype communications management scheme comprised of Registration and Interest agents.

![Diagram](image-url)

**Figure 6-5** Discovery of Other Entities by Interest Agents
6.2.3 Object Mobility

In current DIS simulations, the physical distribution of entities bears no correspondence to their distribution in virtual space. This gives rise to an excessive need for long-distance communications, since it is equally likely that an interaction will take place over the full length of the simulation network as it will over a local link. This has a pronounced effect on the scaling properties of the communications network, which is ameliorated only by a factor through the use of multi-casting to define sub-sets of inter-communicating entities.

One way of reducing this problem would be for the entities themselves to migrate over the processing network to mirror the real-life motions in virtual space of the objects they represent, as in Chapter 5. We have already seen in section 6.2.2 above how similar motions of entity agents can help to localise decision making on the setting up of communications links. Were the entities themselves to move, it would have the effect of forming local clusters with the other entities close to them in virtual space. This would shorten the actual communication paths between entities, and between each entity and its registration agent, allowing traffic to be removed from the long-distance network.
The main differences with this scheme, compared to the mobile models in Chapter 5, is that we are here considering models which have requirements for appreciable computing resources that cannot be met everywhere on the network, which will take an appreciable time to migrate due to the size of their state-space, and which have a need for communications with other models over an extended area. The problems are hence considerably more complex.

The requirements for entity migration are in many respects similar to those for the motion of the entity agents. All of the six criteria given above will apply, but there are several additional considerations:

7) there should be no significant interruption in the operation of the entity due to migration,
8) the communications load caused by entity migration should not dominate the total communications, either as an average through time, or as a transient peak,
9) the current state of the entity must be preserved through the migration,
10) the entity must resume operation using computing resources which can support it adequately, both hardware and software,
11) the entity must not lose contact with other resources which it is calling upon, e.g. databases, special hardware or human operators,
12) an entity which has been constructed out of a number of object components must have its component structure preserved. This will frequently mean that most or all of its components should also migrate, and that these requirements will need to be applied recursively throughout its component hierarchy,
13) the algorithm controlling the mobility of entities must be stable. There is more than one sense in which its stability can be assessed. The most important criterion is that (13a): for a fixed distribution of entities in virtual space and a fixed set of processing resources, the distribution of models over the physical processing network will eventually settle to a static configuration. Another sense is that (13b): any changes in the entity distribution in virtual space will result in changes to entity distribution over the physical network which are bounded in size.
The combined effect of these requirements is to make entity migration a far more difficult concept to implement than entity agents. For many models the requirements cannot be met, and these will have to be regarded as not being suitable candidates for mobility. This may be because the object's state information is too large, or too diffusely defined, to be transmitted over a network in reasonable time, or because it would not be possible to retain control by a human operator over its operation, or because it or its components could not be supported by the hardware/software combinations available elsewhere in the network. In cases where many models are non-mobile, the improvements in performance obtained by allowing the remainder to move are diminished, but in the worst case, performance should be no worse than it would have been without the application of mobility.

None the less, in many large-scale simulations there will be a significant proportion of entities for which mobility is possible, largely those coming under the heading of Computer Generated Forces (CGF, see Cox et al., 1994). The constraints on this mobility are likely to be more severe than on the mobility of an entity agent, since:

- only a limited number of sites would be capable of running any one particular type of model,
- each site will have a restriction on the number of models which it can run while still being able to execute them all in real time.

These constraints result in the need for a more complex algorithm to support entity migration.

In the outline agent design being developed here, model mobility would be accomplished by the addition of two further generic classes of agent. The "Site Search Agent" is launched by an entity's registration agent to look for alternative simulators which have the ability to run the particular type of model required. When a potential site is found, it is assessed for suitability based on the improvement in communications which would result from using it. This assessment makes use of the information contained in the lists of interested entities which have been built up by the registration agent.
Site search agents find vacant slots for models:

Models distributed in virtual space

Figure 6-7 Site Search Agents for Dynamic Model Relocation

Once a decision has been taken to move the site of execution of a model, its current site launches a "Mobility Agent" to carry the current state of the model to its new site with minimum staleness, and to start it executing. The original site continues to monitor the correct execution of the new model for a period, and if it does not run successfully, can send another mobility agent out to kill it off, or to try again. Figure 6-7 shows a schematic view of the operation of a group of site search agents.

This two stage process for managing mobility has the advantages that not only does it respect the constraints on processor capacity needed to run the models, but it has access to information about the current communication requirements for a model (via the registration agent), and so can properly assess the communications implications of a move. It can also be constructed in a way which guarantees the stability and robustness of the overall process of model mobility, points which will be discussed further in the next chapter.

6.2.4 Light-weight models

This category of model refers to objects which are either entities in their own right, or are freely mobile parts of other entities, and which do not have particularly complex behavioural characteristics or dynamics. Models fitting this category can be implemented directly as agents in whatever agent programming language is being used, and allowed to navigate themselves over
the network as if they were registration agents. As their behaviour can be interpreted wherever agent code is supported, there are fewer restrictions on the location of these models than for the more complex and resource-hungry ones more typical of military simulation. Examples of light-weight models lacking sensors would be balloons, parachutists, decoy flares and buoys.

This concept of the light-weight model can be viewed as a development of the mobile-models scheme from Chapter 5, with the improvement that these models can now participate fully in the sensor modelling arrangements included in the new agent architecture. The presence of a light-weight entity can be discovered through interaction of another entity's interest agent either with the light-weight entity directly, or with any attention agents it may issue. More complex light-weight models may have their own sensing arrangements, and issue interest agents on their own behalf. This would allow the modelling of such things as 'smart' or guided munitions as light-weight models.

6.2.5 Environmental modelling

Various aspects of the environment surrounding the entities in a simulation have a considerable influence over their operation. These aspects may include such things as visibility (in various wavebands), temperature, illumination, and the features of the terrain over which the entities are operating, both natural and man-made. For example, the decision as to whether a particular target is visible in an infra-red band involves knowledge of its position, size, temperature distribution, the apparent temperature of the terrain forming its background, atmospheric transmission, and the existence of a line of sight between target and sensor over the terrain between them, as well as the sensor characteristics.

In a conventional distributed simulation, the description of the relevant parts of this environment is duplicated in each simulator. The information duplicated must cover all areas of the battlefield which that simulator is likely to encounter. This duplication is a source of several problems in current simulations:

- it is very difficult to ensure that the data is consistent between different simulators, particularly as they may be using different formats in which to store it,
- as battle-areas increase in size, the amount of data required becomes impractically large to store at every site, i.e. distribution is not a scaleable solution,
- worst of all, it is very difficult to alter the environmental data in a consistent way to reflect the changes caused by the actions of entities, and so allow entities to interact through the medium of the environment.

One way to solve all of these problems is to construct a fully distributed environmental model, where each section of the model is represented once only on a section of network responsible for its maintenance. The concept of distributing an environmental model in this way was originally proposed by Sapaty (et. al., 1995a, p203) in the context of dynamic modelling of terrain.

Consistency is guaranteed by only having one master copy of each section of the environment. The model can be scaled up to accommodate larger battle areas merely by expanding the network, provided that no difficulties of access are thereby introduced. Finally, it should be possible to introduce changes into this model during the course of a simulation to reflect the actions of entities upon the environment, and for these changes to influence other entities. The main requirements for this model are:

1) the facilities should be generic, making no assumptions about the nature of the environment being modelled, or the manner in which entities will interact with it,

2) the environmental modelling must allow of dynamic processing, so that autonomous processes such as drifting of smoke, and movement of water or soil can be adequately represented,

3) entities must be able to obtain environmental information for their own position, and for any other positions within reasonable distance of them, so that, for example, intervisibility calculations from one entity to another can be performed.

4) entities must be able to influence and alter the environment in any appropriate way, both by discrete actions, and by more continuous processes.

5) The environmental models should have the characteristic of a seamless whole, despite being distributed in sections between the different nodes of a network. The access mechanisms should be able to obtain information which extends beyond the boundaries of a single section. Similarly, the mechanisms for influencing the environment should be able to cause effects which are not limited in scale to a single section, but will automatically spread to whatever extent is needed.
An entity should be able to obtain a temporary copy of the information contained in one part of the environmental model, and thereafter automatically receive any updates to that model for as long as it remains a registered copy holder.

The final requirement may seem to be departing somewhat from the principle of having only one version of the environment for any area. The requirement is included to allow human operators to be provided with the outside world views which they need to control their entities. These views are currently produced by high-speed graphics generators, local to each simulator, which need access at high bandwidth to the information about the section of terrain to be displayed. A local copy obtained a little time in advance is the only practical way of achieving this within current network bandwidth constraints.

The presence of these local copies gives rise to a potential problem with consistency. If models were permitted to make environmental changes directly on their local copy before transmitting them to the master copy for dissemination, then incompatible changes could be made by different sources. This would give rise to conflicts which would need to be resolved by the master copy manager, and might result in the need to reverse the changes to the local copies already made, leading to credibility problems with the human operators. To prevent this, all environmental changes should be handled in the first place by the master copy manager.

These requirements will be met by specialised classes of "Environmental Agents". These agents are responsible for maintaining models of the environment for the areas within which the entities are operating, as it affects the functions of the sensors. The complexity of the environmental agents will vary widely depending on their functions, and the degree to which entities may interact through the medium of the environment, for example:

- an illumination agent need be relatively simple, depending only on time of day and cloud cover, with no need to react to the behaviour of the entities,

- an atmospheric transmission agent would be more complex, since it would take account of emissions of smoke etc. from the entities in its neighbourhood. It would also need to simulate the dispersal of these emissions with convection and wind, requiring it to interact with the transmission agents for the areas of battlefield around it as the smoke
drifts. Finally, the interest agents of entities in the vicinity must have access to its transmission model to assess their detection ranges,

the most complex of all would be the agents responsible for the terrain model, since these must allow entities and weather to cause arbitrary changes to the terrain, which may spread well beyond the limits of any single area, and modify the behaviour of a large number of other entities. Examples of changes can range from simple craters, foxholes and damaged structures, through extended features like tank tracks and the laying of minefields, to major changes such as landslides, floods and road construction.

6.2.6 Situational Analysis

In a distributed and decentralised simulation, with no global data exchanges, the on-line analysis of situations as they develop in the battlefield becomes a non-trivial problem. It is no longer possible to have the equivalent of a DIS "Stealth" vehicle, operating at a single site, and able present a view of any part of the battlefield from the information which is passing across the communications network. In the type of simulation being considered here, the data is scattered, its volume is huge, and it could not be broadcast without swamping the network. It will be necessary for any on-line analysis system to actively search out the information which it needs, and collate it to reduce its volume before transmitting it to the point where it is to be used. Sapaty has suggested that a mobile agent technique has considerable attractions in this application (Sapaty et. al., 1995a, p.204, 'roaming cockpit').

Such agents should be able to satisfy a number of requirements:

1) the agents should be able to navigate across the network to search out the sources of the information autonomously, and track these sources through the network if they are themselves mobile.

2) the system should have the flexibility to change the information collection functions by launching new types of agents, without need to recompile code contained in all simulation nodes.

3) a single agent should be able to collect information from several simulation nodes, and combine it to form a partially interpreted picture of the situation as it affects a number of entities.
4) agents should be programmed to recognise specific situations as they develop, using pattern recognition techniques, and alert their owner only when it becomes necessary, operating in silence until they do so.

5) agents should be arranged in a hierarchy, which can form a summary of the situation affecting the entire simulation. Local agents should be aware of what is significant, and extract it at appropriate rates. Medium-scale agents should combine data from the local sources to provide a summary which is sent up the hierarchy, with the requester receiving the net result. Since data would be summarised and collated throughout this process, at no point should an excessive communications loading develop.

Many of these mechanisms can be used to construct models of the information gathering parts of command and control systems. It would be necessary to introduce some errors into the process (e.g. missing, corrupted, late or misinterpreted data) to present a realistic impression of the reliability to be expected of the real system. In this case it might be necessary also to simulate the generation of 'correction reports'.

The author has coined the term "Collation Agent" to describe the specialised classes of agents needed to inform interested parties as to the state of play within the simulation. These may need to be able to track entities as they migrate through the network, assemble status information from the members of co-operating groups, and summarise information over widely differing areas of the battlefield to perform the various functions required within a large-scale simulation. Their designs are certain to be specific, rather than generic, due to the specific nature of the information which they would be required to handle, and the specific algorithms needed to collate it.

6.3 Example Application of the Generic Architecture - The Control of Aggregation

There is considerable benefit to be gained in a large-scale battle simulation from representing a complete group of entities as a single object (or aggregate). The majority of this benefit comes from the reduction in computation needed to represent the behaviour of this group. In particular, when two such aggregated objects engage each other in battle, the outcome can be computed from the statistics of the situation, and presented as a set of attrition rates, with vastly less

105
computation than would be required to model the interaction of all the entities involved individually.

However, problems arise when such aggregated objects need to be used within a synthetic environment which also contains individual entity models. The approach normally adopted when entity models need to interact with an aggregate is to convert the aggregate into a set of new entities, and resume modelling the interactions at the entity level, a process known as deaggregation (or disaggregation).

Deaggregation is no trivial operation, since the new entities created must be given convincing positions and behaviours which have no representation in the parent aggregated model. Successful deaggregation requires knowledge of the operational doctrine of the group, its commander's intentions, details of the environment within which it is operating, and the disposition of other forces nearby.

Artificial intelligence techniques have been applied to approach this problem (Cox et. al. 1995), but it is by no means solved, and is not considered to lie within the scope of this work. Instead consideration will be given to structures which could potentially support the decision making required by deaggregation, by ensuring that the data needed is available, and that the various models involved can be operated successfully. The main requirements for this are as follows:

1) access to the positions of both friendly and opposing forces in the vicinity is needed, so that a decision can be taken as to when it is sensible to commence deaggregation, and so that the new entities created are not positioned in the middle of another force, or on the wrong side of the front line;

2) information is needed about the terrain over which the aggregated entity is spread, including type of ground (forest, lakes, road etc.) and the relative vulnerability of objects occupying it, so that the new entities can be positioned in places which it would have been sensible for them to have reached;

3) the ability to find spare modelling capacity of a type suitable to operate the code for the new entity models being created, and to start them operating there;
4) the ability to create a mixture of entity types, appropriate to the type of aggregated model being represented, and to describe the appearance of each different entity so that they can be displayed on simulation nodes which have not encountered this type of entity before;

5) The ability to control each of the new entity models, so that their individual behaviours are sensible, e.g. vehicles do not collide with each other, but keep in formation or provide each other with cover against attack;

6) the ability to extract information from the entity models as to their current circumstances, summarise it and pass it back to the commander in identical form to the information which was available from the original aggregated models, to help in the formulation of new commands. This information may need to be deliberately corrupted to simulate the effects of imperfect military command and communication systems;

7) the ability to pass fresh commands down to the new entity models, so that they can continue to carry out the commander's plans, however these may evolve with circumstances;

8) the ability to detect when it is no longer necessary to maintain individual entity models. The ability to use the summarised information about the state which these have reached to update the original aggregated model and resume use of it, thereby releasing modelling resources for use by other aggregated models.

This assumes that information about the relevant doctrinal rules needed to command such a group and the knowledge of the commander's current plans are both available within the aggregated model, so that the disposition of the entity force created is militarily reasonable, and it remains capable of continuing to carry out the plan envisaged.

It can be seen from the above that the control of aggregation and deaggregation is a complex task, requiring a wide range of information and capabilities. However, nearly all of the requirements above have already been encountered within the previous categories of agent applications described earlier in this chapter. The registration agent for an aggregated group is in a position to make a decision to commence deaggregation, either from knowledge of the position of the model, or from the information contained in its list of detecting sensors which would
Generic Roles

indicate when the model is required to interact with neighbouring entity models. Two further steps are then needed:

- generation of suitable initial states for the new entity models, and
- locating simulation resources which can run them.

The registration agent has access to the majority of the information needed to make informed choices for the initial states of the new entities. It can find the positions of all the neighbouring entities from their registration agents, and it can obtain the terrain and environmental information needed from the environmental agents controlling its locality. Of course, the mere possession of this information does not make the task any less demanding of intelligent assistance.

Deaggregation need not be an all-or-nothing process, but may take place progressively, entity-by-entity, as recommended by Cox et. al. (1995, p454). The new entities may start life as lightweight models (as defined in section 6.2.4), but at some stage it is likely that a full-scale computer-generated simulation of each will be needed. This is when the site search agents and mobility agents already developed for model mobility can be used to dynamically allocate the entities to unused simulation resources within the network. The control of these individual entity models would be entrusted to command agents (ibid). Note that this scheme, in which a single registration agent is responsible for a number of entity models, is somewhat different from that proposed by Sapaty et. al. (1995a,b), in which a group of mobile agents would model the behaviour of a group of entities by collaborative processes involving exchanges of messages and voting.

While operating these entities, various types of collation agent would be applied to search out and summarise the overall state of the group. This is transmitted back to update the states of the original aggregated model, so that it can continue to pass information to the higher-level simulation of which it was a part. This information could also be used to decide when the individual models were no longer necessary, and to trigger re-aggregation, a process involving little dislocation provided that the aggregated model has been kept properly updated. This would then free simulation resources for use by other deaggregating groups.
One incidental advantage which might be gained from the use of aggregated models within the simulation is a reduction in the number of separate agents needed to perform the interest management function. This assumes that it is possible to continue to manage the interests of an entire group with a single agent after it has deaggregated, as well as when it is aggregated.

6.4 Discussion

In this chapter an architecture for distributed simulation has been proposed which makes extensive use of some existing and several proposed new mobile agents in a number of cooperative roles. The new roles include:

- **Operator Interaction Agents** for control and monitoring of the behaviour of individual entity models from remote sites;
- **Entity Interaction Agents** which may be used to mediate the interactions of closely-coupled models over a limited bandwidth network;
- **Registration Agents** which perform a geographically mapped sorting of the entities in a large simulation;
- **Attention and Interest Agents** which, in conjunction with registration agents, perform the process by which entities discover the existence of other entities of relevance to them;
- **Site Search and Mobility Agents**, which can be used to allocate entity models to suitable free computing resources when they are created, and subsequently re-allocate them dynamically should it be possible and advantageous to do so.

The existing roles were:

- **Environmental Agents** of various sorts responsible for maintaining a distributed model of the physical environment in which the entities are operating, and accounting for indirect interactions between entities which may occur through the medium of this environment;
- **Collation Agents** which gather and summarise information about various aspects of the progress of the simulation, either for use in later analysis, or for immediate use by entity models operating at command level.
By its very nature, this architecture must be regarded as highly speculative. It was intended as a vehicle for raising questions of feasibility that could be answered by further research, rather than settling questions of itself. In subsequent work some of the agent roles defined above have been prototyped in order to begin to establish whether some aspects this approach might prove to be feasible.

One of the main advantages displayed by this approach is that it gets away from the total reliance on the standard communication protocols exhibited by current distributed simulation systems. Many of the model-agent and inter-agent communications are purely private and can contain any information needed to perform the functions required. Since both communicating elements originate from the same site and designer, no formal agreements with other organisations are needed to modify these functions, provided that the integrity and robustness of the overall system are not compromised thereby.

Systems constructed using self-contained and private inter-agent protocols could be considered to be an extension of the component architectures, discussed in Chapter 2, further into the realm of distributed operation. An analogy could be drawn between a single component module operating on one node of a network, and the complete set of agents obeying one of these private protocols. Each of these is a replaceable element of the overall computational system, with the difference that the set of agents possesses many more degrees of freedom with respect to the information it can gather and the influence it can have on the rest of the system.

Much further work remains to be done to validate these concepts, particularly in the areas of load balancing, stability, and robustness. Specific examples of simulation agents need to be implemented to explore the extent to which the generic agent roles discussed here can be applied to real modelling problems, and to determine the detailed performance requirements for various types of simulation. Work has started on the prototyping of some of the agent types for use in a simulation context. In Chapter 7 a prototype system of Registration and Interest agents is described, implemented using the WAVE language (see Chapter 2). In Chapter 8 an example of an Operator Interaction agent is given, making use of the Java language.
Chapter Seven
Chapter 7 Design and Prototyping Work on Generic Agent Roles

7.1 Introduction

This chapter goes into more details of the design aspects of some of the generic agent roles defined in chapter 6, concentrating on the roles intended for management of communications and computing resources; the operator interaction role will be considered in more detail in chapter 8.

Several of the requirements listed in Chapter 6 give rise to potential implementation problems, particularly in the areas of load balancing, robustness and stability. The main purpose of this chapter is to present an outline of how these problems can be approached within the above architecture.

Section 7.2 considers the registration, interest and attention agent roles, intended to control the communications between models according to their needs for data. Section 7.3 investigates the agent roles needed to manage computing resources using entity mobility, the site search and mobility agents, which require some extension to the role of the registration agent. In section 7.4 a prototype implementation of a system comprising registration and interest agents is described, and a model predicting the performance and scaleability of this system is developed in section 7.5. Section 7.6 contains a summary and critique of the work in the chapter.

7.2 Design Aspects of Communications Management

The major requirements for the communications management roles were stated in 6.2.2 as being the need for efficiency, scaleability, genericity, robustness, topological invariance and immunity to changes in entity distribution. In this section the proposed agent architecture will be examined in the light of these requirements, and elaborated where necessary in an attempt to satisfy them.

7.2.1 Efficiency

The communications efficiency of any scheme of communications management is largely determined by the degree of selectivity which can be achieved, i.e. the number of unnecessary data transmissions and receptions which can be eliminated. The DIS architecture makes little attempt to be efficient in this respect, since it depends on regular broadcasts of all the status information pertaining to every entity, to be received by all other entities. The HLA scheme (DMSO, 1995) aims to do considerably better than this by requiring all entities to declare which
classes of model they need data from, and over what geographical area. This information would be used to select the membership of various groups which receive sub-sets of the information available by means of a multi-casting communication scheme. In addition, the run-time infrastructure of HLA maintains a register of the existence of the entities, and their relationship to each other, eliminating the need for regular broadcasts of this information.

The agent scheme described in Chapter 6 aims to be at least as efficient as HLA, and has two advantages which may result in even higher efficiency. The first is that the selectivity achieved by a system of interest and attention agents is likely to be considerably higher than that achieved by a simple geographical limit, since approximations to actual sensing algorithms can be employed to eliminate undetectable entities before any data exchange has occurred between them. The second advantage concerns the ability to tailor the nature of the communications link to the requirements of the receiver by, for example, reducing the data-rate where possible. This would be of particular benefit when simulating very wide-area sensing systems such as warning radars, which would otherwise need to belong to every multi-cast group in the simulation, and be swamped with data as a consequence.

Against these benefits must be weighed the additional data transmissions needed between an entity and its registration agent and the processing requirements of the agents themselves. The overhead introduced by the former should be relatively insignificant, since they in effect add one to the list of entities being communicated with. The latter aspect will be considered further in the performance model introduced in section 7.5.

7.2.2 Scaleability

The scheme certainly satisfies one of the prime pre-requisites for being scaleable in that there is no central management resource, since the agents are self-navigating, and the servers respond entirely to their local conditions. This lack of a central failure point also provides one reason for believing that the system can be made fault-tolerant, though other elements of fault-tolerance will be needed. The system should be scaleable as far as the duties of the servers and agents are concerned - no part of their work depends on the scale of the entire simulation, as will be demonstrated in more detail in section 7.5. However, this is not sufficient to ensure the scaleability of the overall system. The capacity and connectivity of the underlying communications network supporting its operation must also be such as to allow agents and their
originators to maintain communication for any reasonable distribution of agents. Section 7.3 on entity migration will discuss ways of reducing some of the problems implicit in this requirement.

7.2.3 Generic Properties

This aspect of the design is concerned with the extent to which the various components making up the system will need to be adapted for use with different simulation models. If every part of the system needs to be re-designed so that it can manipulate specialised data structures, or call upon specialised operations of particular simulation models, then, although the various agents can be said to be fulfilling their generic roles, they cannot be said to be part of a generic design. On the other hand, if one of these agents can be implemented directly in a generic form, with minimal need for specialisation for use with particular models, then its design can be said to be generic.

The ability to make use of generic designs results in a considerable saving in cost and effort through reusability and standardisation, but it is unlikely to be applicable to all of the agent roles outlined here. An example of a generic design for an operator interaction agent will be described in Chapter 8, in which the specialisation needed for particular applications can be achieved through the use of configuration files, or by extending the generic design by means of inheritance.

Preliminary indications from the prototyping exercise in Section 7.4 are that it may be possible to implement a registration agent directly in generic form, since it mainly deals with identities and positions of entities, rather than with any more detailed information about them. Similarly, the functions performed at a node of the geographically mapped server network may also be considered to be generic. Interest agents and attention agents are very unlikely to be generic, due to the specialised nature of the sensing algorithms they approximate and of the signature data they carry respectively. Specific designs would need to be developed for each type of detection interaction, though it may be possible to avoid producing new designs for each individual sensor type by suitable parameterisation of these.

7.2.4 Invariance to Network Topology

The exact topology of the server network should have little influence on the behaviour of the system, provided that it is a sufficiently well-connected mesh for agents to be able to navigate
blindly over it, optimising their positions purely with regard to the local coordinates of the servers. This condition implies that there be no local minima in the position optimisation process. If an agent were searching to adjacent server nodes only, this would require that for \( n \) dimensions of search space, there should be at least \( n+1 \) links from each node to others, and that the set of coordinate vectors along those links should span the search space, i.e. the dimensionality implied in the connectivity of the multi-processor network must be at least as great as that of the real-world space being simulated.

In practice, local deviations from these ideal conditions could be tolerated, provided that the agents were allowed to search through more than one link to establish the optimality of their position, i.e. that an agent at a sub-optimal location should always be able to find another location which would improve that position within no more than 2-3 links. This ability provides another element of fault-tolerance, since local minima could appear briefly when individual links or servers fail, or a server migrates to balance load (see 7.2.6 below).

The network topology could have a bearing on scaleability. If one part of the network had relatively few links and nodes, then a large concentration of agents trying to navigate through this region could cause local overloading. This can be prevented by ensuring good overall connectivity within the network.

7.2.5 Fault tolerance

Some aspects of fault tolerance have already been touched upon - the absence of any central management structure which would give a single point of failure for the whole system, and the ability of the agents to continue navigating despite local distortions of the position coordinates caused by failures in the server distribution. Another aspect concerns the agents themselves. The system should be able to survive and recover from the failure of one or more agents. As no identifiable single point of failure exists, ensuring the survival of the overall system should not present a great problem, since the failure of any one agent will have little impact on the operation of the entire system.

When an entity's registration agent fails, the only effect will be to cause that entity to have its communication interests frozen. The existing interest list would remain unchanged, as no new discovery operations would be performed. To recover, the entity needs to be able to recognise
that this has happened and launch a fresh registration agent to replace it. Since entity agents and their originators need to keep in regular contact, detecting the loss of an agent should not prove difficult.

Similarly, the loss of an interest or attention agent should not cause more than a temporary interruption in the discovery process in one part of the network, since these agents are emitted at regular intervals by the registration agents. Further robustness features of the design can be seen in Tables 7-1 and 7-2 and Appendix F.

7.2.6 Load Balancing

Finally, one major aspect for which elaboration of the proposed design will be needed is the question of load balancing (requirement 3). A serious deficiency of the scheme described in Chapter 6 is its lack of any way to respond to changes in the distribution of models in virtual space. During battlefield simulation, it is usual for entities to congregate together in large groups, particularly where engagements are occurring. If nothing were done to prevent it, this would result in large concentrations of entity agents on certain servers, which would be seriously slowed as a result.

The modification proposed to overcome this involves allowing servers to change their area of interest, and placing a limit on the number of entity agents which can occupy any one server. When a server becomes full, the agents it cannot accommodate will cluster around it on adjacent servers who will be able to detect this situation from the fact that the majority of their clients' areas of interest do not correspond with their own, but match the adjoining area. Under these circumstances, these peripheral servers will slowly alter their area of interest to overlap with the overcrowded area, strengthening its resources of servers.
<table>
<thead>
<tr>
<th>Simulator Interface</th>
<th>Nodal Processing</th>
<th>Registration Agent</th>
<th>Interest Agent</th>
<th>Environmental Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Launch a Registration Agent for a local entity.</td>
<td>Read centre positions and times of neighbouring nodes. Calculate new centre position and time for this node.</td>
<td>After launch, navigate to best position in nodal network.</td>
<td></td>
<td>Navigate to position in network nearest to own area of coverage.</td>
</tr>
<tr>
<td></td>
<td>Read entity positions and time stamps of any registration agents</td>
<td>On arrival at any node, puts out the position of its entity and a time stamp.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send information about environmental effects caused by this entity.</td>
<td>Receive environmental effects caused by entity - send to environmental agent</td>
<td>Periodically spawn a new Interest Agent.</td>
<td>Move out, to find other registered entities for interaction.</td>
<td>Collect any environmental effects caused by entities and update information.</td>
</tr>
<tr>
<td>Send an update of position and signature information for entity.</td>
<td>Maintain a set of position and signature information for own entity.</td>
<td></td>
<td>Estimate whether entity might have been detected, using signature and environmental information.</td>
<td>Publish updates to environmental information for node and environs.</td>
</tr>
<tr>
<td>Receive a list of interested entities.</td>
<td>Accumulate a list of interested entities and send to originator. Accumulate another list of entities sending data to this one.</td>
<td></td>
<td>Inform own and other entity’s Registration Agent of the successful detection</td>
<td></td>
</tr>
<tr>
<td>Send data from entity out to other interested entities, at the requested update rates.</td>
<td>Robustness: Check on continued operation of own Registration Agent - launch a new one if necessary.</td>
<td>Robustness: If neighbouring nodes or links are missing, reconstruct them.</td>
<td>Robustness: If own entity model stops, inform an operator.</td>
<td>Robustness: Check on existence of neighbouring environmental agents. If missing, create new one, with duplicate information.</td>
</tr>
</tbody>
</table>

Table 7-1 Main Interactions between Registration, Interest and Environmental Agents
<table>
<thead>
<tr>
<th>Simulation Set-up</th>
<th>Simulator Interface</th>
<th>Registration Agent</th>
<th>Site Search Agent</th>
<th>Mobility Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send out a Registration Agent in advance of entity creation.</td>
<td>Publish information about models available and spare processing capacity.</td>
<td>Periodically spawn a Site Search Agent to find new sites.</td>
<td>Spread out, looking for sites with similar model to originating one.</td>
<td></td>
</tr>
<tr>
<td>Receive location of initial site for entity from Registration Agent.</td>
<td>Receive instruction to move an existing entity to a new site from Registration Agent.</td>
<td>Using lists of transmitting and receiving entities, estimate communications costs of candidate sites and current site. Choose best site.</td>
<td>Send position and degree of suitability of candidate site to Registration Agent.</td>
<td></td>
</tr>
<tr>
<td>Dispatch Migration Agent to initial site with start-up data.</td>
<td>Dispatch Migration Agent to new site with start data.</td>
<td>(At new site) receive migration agent and start executing new entity with its state information.</td>
<td></td>
<td>Travel to new site and deliver information needed to start new entity.</td>
</tr>
<tr>
<td>Robustness: check that instance starts operating - relaunch on same or alternative site if not.</td>
<td>Robustness: monitor performance of model on new site compared with old. Revert to old site if a divergence is found.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7-2 Main Interactions of Site Search and Mobility Agents
Any entity agents which were correctly positioned before this interest alteration will be free to re-optimise their positions with respect to the server network, and would migrate outwards, relieving congestion in the overcrowded area.

When the entities causing the overcrowding have moved away, the concentrated server network will need to expand back to its original distribution in virtual space. Each server could do this using local information by moving its area of interest so as to equalise the interval between itself and its neighbours on all sides. A trial implementation of this load balancing scheme is included in the prototyping exercise described in section 7.4.

It should be noted that this scheme can at best only partially alleviate the loading effects of an inequitable entity distribution. Even though the total number of entity agents on each server is bounded, each agent is inevitably having to do more work, since the potential number of interests in its neighbourhood has increased with the concentration of other entities. In the absence of any load-balancing this would result in a square-law increase in loading with entity concentration. The server overlap scheme described above should reduce this to an approximately linear increase, and the capacity of the server network still needs to be planned to accommodate the maximum entity concentrations which can reasonably be expected in the course of a realistic simulation.

### 7.3 Design Aspects of Resource Management through Entity Mobility

Section 6.2.3 described a scheme for reducing the amount of long-distance communication needed in a distributed simulation by encouraging the dynamic re-location of suitable entity models. Since these models in general required significant processing power and other resources to execute, this problem cannot be regarded as a purely communications management issue, but must be tackled in a broader context. A possible algorithm for controlling object migration will be proposed in this section.

Each potentially mobile entity will periodically send out a site search agent which has the responsibility of finding candidate sites where the object may execute. This agent will split into a number of parallel threads, each of which sends back information on the site it has found. The object ranks these according to criteria discussed below, and attempts to migrate to the best of them, if any are significantly better than its current site. The ranking criteria to be used for this
Design and Prototyping Work on Generic Agent Roles

need to be chosen with considerable care if the requirements are to be met. They must take into account the physical distribution of models, the potential processor loading, and the potential communications loading resulting from the migration.

Certain criteria will be regarded as absolute, and a site will not be treated as a candidate unless they are met. These are:

- the site must have an execution environment suitable for the model to run in. In practice, this will usually mean that the site has at least a compiled copy of the model's dynamics, or a generic model dynamic which can be parameterised to approximate the behaviour expected of the model, and copies of any data resources which it needs;
- the site must have sufficient spare processing capacity to execute the model at the rate required;
- the area of interest at the new site must be closer by a given factor to the model's position in virtual space than its existing site;
- the cost of communicating with the object in its new site must be significantly less than in its existing site (see below).

Having met these absolute criteria, the criterion used for ranking against other possible sites would be a weighted sum of:

- the closeness of the site's area of interest to object's virtual position, and
- the cost of the communications loading for the object in the new site.

The first of these is simple to calculate; the second requires the establishment of a metric bearing some relationship to the problems of communicating with the object in its new site.

This communications cost metric is central to the success of the object mobility algorithm, and should include factors such as: the proportion of long-haul communications versus short links, the use of particular links which may be already heavily loaded, etc., and must be based on knowledge of the physical positions of the other entities currently in contact with this one.

There remains the question of how best to organise the computation of this communication cost. The suggestion proposed here is to extend the capability of the registration agent used to set up
the communication links, so that it accumulates the cost of each link at the same time. This would ensure that the communication cost for the current site would always be available. When assessing other sites, a similar agent would be launched which would measure the potential communication cost from the new site, but not actually set up any new links.

This scheme will now be discussed in relation to its compliance with the requirements set out in section 6.2.3.

7.3.1 Stability

The major rationale behind the use of two criteria is to ensure the stability of the algorithm. Although the main purpose of object mobility is the reduction in communications loading, use of that criterion on its own could not be guaranteed to produce a stable system. To see this, consider the case where two mobile objects are in communication, and both actively seeking a new site at the same time. Each would find it beneficial to move closer to the other to reduce the length of the link between them. Each would make its move independently of the other's move, giving a high probability that the models would simply swap sites, without actually becoming any closer. This oscillatory behaviour could continue indefinitely.

The introduction of a geographical criterion requiring that the object move closer to its position in virtual space, as well as reducing its total communications cost, has the effect of removing the possibility of setting up indefinite oscillations in the physical positions of objects. Each object has an end point to its migration which it must converge towards, reducing the magnitude of the error by at least a fraction $f$ at each move. If the positional error vector is $p$, then after a move the new vector $p'$ satisfies:

$$|p'| \leq (1 - f) |p|$$

This constrains the object to lie within a hyper-sphere of diminishing radius centred on the correct virtual position. Since there is only a discrete set of sites capable of executing the model of which this object is an instance, after a finite number of moves this sphere must eventually reach a size such that it contains at most one possible site for the object. If it contains one, this may become its final site. If it contains none, the object is already on its final site.

Thus, the migration of each object is stable in the sense required by condition 13a, for a fixed distribution of objects and sites in virtual space. This convergent behaviour is completely
independent of the presence of other objects in the environment, though these other objects will influence the timing of migration, and the sequence of sites visited, through the communications cost part of the criterion.

The other aspect of stability, requirement 13b, was that the migrations caused by changes in virtual position of objects (or, by extension, movement in the interest points of sites) should be bounded in extent. If the object itself moves by a vector $\mathbf{D}$ in virtual space, its next migration cannot exceed a distance of $M$ where:

$$M \leq (2 - f) \cdot (|p| + |D|),$$

since $M = |p - p' + D|$

and $|p'| \leq (1 - f) \cdot |p + D|$

which is clearly bounded. An identical bound applies if, instead of the object's virtual position moving, one considers the centre of interest of its existing site as moving by a vector $-\mathbf{D}$ in virtual space. In addition, the influence of movements by other objects is already covered by the considerations in the previous paragraph.

7.3.2 Accommodating Component Structures

An entity may consist of several component objects, possibly executing on different computers within the network, but logically connected and interacting to emulate the behaviour expected from the overall entity. This concept of a constructed entity can be extended, as is done in MulTiSIM (Chapter 3), to include any linked grouping of objects, for example a squadron of aircraft or a tank battle-group.

When considering the mobility of such constructed entities, it is perfectly valid to follow the algorithm given above for each component independently to find it a suitable new site for execution. The inter-object communications paths within the entity would be treated as one of the contributing factors within the communication cost of the component object. Treating the components independently in this way is sufficient to permit migration, but may be unnecessarily strict, since the communications paths with the other components would be extended by such a move. In practice, this might make it rather difficult to justify moving any single component.
Design and Prototyping Work on Generic Agent Roles

A more satisfactory algorithm would consider the communication costs resulting from a combined move of all components, or as many as are suitable for migration. The inter-component links resulting from this would be kept short, so that the change in communications cost would be dominated by the interactions with other entities. Unfortunately, it is no longer a trivial matter to design an algorithm which can find the optimum distribution of several components simultaneously, when the relative placement of each interacting pair of components affects the outcome of the cost function.

If there are n components, each of which has m likely sites, the number of possible configurations rises to m^n, and each of these configurations has a communication cost which could have as many as n.(n-1)/2 inter-component contributions to evaluate. An entity with only 5 components and 3 sites for each could require 2,430 cost terms, combining n.(n-1).m^2/2 = 90 independent cost evaluations into 243 different cost values. In its general form, such an algorithm would clearly be impractical for implementation within a real-time system. The most practical approach would be to adopt some restriction in the nature of the permitted solutions to this problem to simplify the algorithm.

One obvious restriction would be to insist that all the objects being considered for a combined migration should execute on a single site. Their inter-component communication costs could then be taken as effectively zero, and only the m different sites need be considered. However, this is a severe restriction on the nature of the components, and gives no additional capability beyond the approach of considering the entity as a single object. A less restrictive simplification would be to avoid considering all possible combinations of component positions by partitioning the sites into a small number of arbitrary groupings, based on proximity, each grouping having just one possible site for each component in the entity. In the example above, there would be just three candidate groupings, each with ten inter-component links to evaluate. This would not find the optimal distribution of components, but could permit reasonable sub-optimal solutions to be found and adopted.

7.3.3 Run-Time Resources

Requirements 10 and 11 are closely related, as they both concern the computing and information resources being used by the model. If the model is to continue to operate successfully after a migration, then all the resources it is using must either be available at its new site, or from its
original site via communications links. This consideration applies equally to outputs of information being provided by the model for use by other components of the simulation. The resources and outputs could include:

- compiled code and computing environment needed to implement model behaviour,
- data-bases covering such things as threat libraries, terrain models etc.,
- controlling inputs from operators,
- status and view-point displays required by operators,
- connections to hardware rigs.

How difficult it would be to provide compiled code for a model at a remote site depends to a large extent on the degree of standardisation of the set of models in use. Many simulator models are unique, designed by specialists for a specific purpose, and could not expect to execute elsewhere than at their home site except by special pre-arrangement. Other models, such as those to be found in computer generated forces or frameworks such as MulTISIM, have more standardised requirements which would allow them to operate wherever their parent framework has been established. It is this latter type of model that a mobility algorithm is intended to support. An object-based modelling structure, in which the set of attributes for each instance is held in a well-defined and self-contained data-structure, is also of benefit in this application since it allows the state-data of each instance to be readily extracted and duplicated.

A reasonably efficient language supporting code mobility would be of considerable help in increasing the range of models which could be rendered mobile. The most obvious candidate for this is Java, for which ‘just in time’ compilers, capable of producing efficient native code from the mobile Java byte-code, are becoming available. Problems of transmission time would be exacerbated, as discussed in the next subsection.

The problem of access to data-bases is also likely to restrict the classes of model which are candidates for mobility. Unless a data-base can be duplicated at a number of sites before a run, or is small enough to be transmitted with the model as it migrates, then the model will need to maintain communication with it over the network, adding considerably to the cost of communication, and increasing the time-latency of the model’s responses. Duplication of data-
bases is not in general to be encouraged, since it then becomes very difficult to modify their contents during the simulation and ensure consistency between the various duplicates. In the military field, many data-bases relating to threats and countermeasures are secure, and could not safely be copied.

Retaining access to human operators or hardware devices associated with an entity poses the most severe performance problems. If the entity has migrated to a remote site, the round-trip time latency for control inputs becomes an issue. Human operators performing tightly-coupled control tasks can readily perceive latencies as short as 50 msec, and any more than 100 msec can result in loss of control during demanding manoeuvres. Worse than this, hardware-in-the-loop tests can easily involve higher bandwidths than the human operator presents, particularly where fly-by-wire sensors or actuators are in use. With current computing hardware and communications systems, single trip times of less than 50 msec are difficult to achieve, resulting in round trip times of at least 100 msec, when the time taken to process information at both ends is included.

From this consideration alone, it would not appear feasible at present to allow migration of those parts of an entity dealing with the dynamics of an operator controlled vehicle. Other types of model with relatively low-bandwidth communications to the operator could be good candidates for migration, however, particularly if they have a heavy communications load with other entities. Many information extraction systems have this characteristic, as will be discussed below.

7.3.4 Object State Transfer

In Section 6.2.3, requirements 7, 8 & 9 all relate to the transfer of object state during migration, and ask that it be complete, timely and economical of bandwidth. These three requirements are closely interrelated, since the time taken to transfer state depends on the amount of data needed to specify the state of an object, and the spare bandwidth available to transfer it. If the maximum time allowed for an interruption in model operation is $t$, then:

$$ t > 8 \frac{D}{B}, \quad \text{or} \quad D < t \frac{B}{8} $$

where $D$ is the quantity of data (Kbytes) and $B$ the spare bandwidth (Kbits/s).
The bandwidth and time-interruption limits will thus set an upper bound on the maximum complexity of any model which can be regarded as a candidate for migration. For example, if a 0.2 sec interruption can be tolerated, and a spare bandwidth of 64Kbits/sec is available, then only 1.6Kbytes of information per object can be transferred (400 floats). Relatively simple computer-generated forces models may only need a few Kbytes to describe their state, whereas models involving more sophisticated KBS reasoning systems to control their behaviour may have knowledge networks stretching to several hundreds of Kbytes. As higher bandwidth networks become more common (typically 155 Mbits/s with Asynchronous Transfer Mode communications), then a much wider variety of models could be open to migration.

The overall time-delay involved in starting a model on a new site may involve more than just the data-transfer time. The time taken to launch a new modelling process can be significant, but this may be done before initiating the data-transfer, and so need not contribute to the overall interruption time. There is no reason to completely freeze the model state during the transfer. Dead-reckoning techniques similar to those currently used in DIS protocols could be employed by the registration or mobility agent to update the states of the model at its new site to partially account for the transfer time. The model would not be able to respond to new incoming information during this time, however.

Looking back at the remaining requirements inherited from Section 6.2.2, we see that it should be generic, scaleable, fault tolerant, give good processor loading characteristics and be unaffected by network topology. The mobility algorithm described would appear to be generic, since it can be described purely in terms of generalised object position coordinates, and communications costs. Any particular implementation of this algorithm would need to specify the type of position data (as must also be done for the relevant filtering algorithm), the method of estimating communications cost and some means of identifying sites capable of running the specific models which are candidates for mobility.

This latter function could best be done using semantic matching within a data dictionary specific to the simulation being undertaken. Exact matches may not always be possible; an object may need to migrate to a site where the models available do not have precisely the same behaviour as its original. However, provided that they are acceptably close, it may be possible for the agent controlling the migration to translate the state information appropriately to allow the resumption
of modelling with a good approximation to the original characteristics. The success of this matching process would be monitored and verified by the original model, as described below under fault tolerance.

### 7.3.5 Scaleability

The algorithm uses no central resources, and can be performed purely by the actions of agents, navigating over the processing network in a very similar manner to the relevance filtering algorithm above. Provided that these agents do not concentrate their actions on too restricted a part of the network, there is no reason to think that this algorithm is any less scaleable than the relevance filtering one. To ensure that such concentration does not occur, it is necessary to ensure that a reasonable number of alternative sites are available for each class of mobile object, and that this number scales with the number of instances of that class in the overall simulation.

The geographical distribution of these sites should also be kept as even as possible to ensure good coverage of the battle arena, but apart from this there will be no more dependency on network topology that there was for the relevance filtering algorithm. The processor loading aspect has been dealt with by ensuring that each processor can reject new models if it is already heavily subscribed. The site search agents will find the nearest alternative site which has spare capacity, while still giving favourable communications costs.

### 7.3.6 Fault tolerance

Again, there are several aspects to fault tolerance as it affects the mobility algorithm. There is no central resource in use which could cause failure of the entire system, so failures would be confined to the effects on individual models, agents and processing nodes. The effects of links or nodes on the navigation of agents would be identical to the considerations given above for the interest management system. If an agent searching for a new site fails, its originating model will eventually issue another one.

The main new cause of potential failure relates to the actual migration of the model. Were the model to fail to resume execution at its new site for any reason, the object being modelled would vanish from the simulation, doing considerable harm to the reality of the simulation in the process. If the modelling resources selected at the new site were to prove inaccurate in their
prediction of its behaviour, then the immediate effect would be less obvious, but in the longer term could invalidate the results of the simulation experiment.

The best way of detecting and recovering from such events is for the original modelling site to continue monitoring the execution of the new model for some period after migration, and if it fails to observe correct operation, abort the new models and resume control over the model's state. It could do this to any degree of accuracy required by continuing to shadow the behaviour of the model, and verifying its own state evolution against that of the new model. In this case, failure of the mobility algorithm for one model would not cause loss to the simulation, since the object could continue to execute from its existing site, but with a less optimised communications loading than if mobility had been operating.

From the design considerations discussed in the preceding two sections, it is possible to begin to derive the main functional requirements for the various agent roles. These have been described in pseudo-code form in Appendix F, and are summarised in tables 7-1 and 7-2 above. In these tables, each column contains the various actions to be performed within a single generic agent role. Reading across a row shows how each of these actions may influence the actions of another type of agent, so that the collective interaction of all of the roles combine to produce the system behaviour needed.

7.3.7 Light-weight Models

The light-weight models described in the requirements chapter can be implemented largely by using the facilities already available from a generic registration agent, and adding specific code to it to perform the actions required of the model. There is little which can therefore be said about these in a chapter devoted to generic design, except to note that in deciding to adopt the light-weight approach for any particular model there would be a trade-off between an increase in the processing load needed to execute the model behaviour in the same processor as the agent, and the removal of long-distance communications needed between the registration agent and the heavy-weight model which it would otherwise represent.

The direction of this trade-off would depend critically on the execution speed of the language in which the light-weight models were implemented. The majority of agent languages are interpreted, and not noted for speed. It would be interesting to see whether a partially compiled
approach, such as that adopted by the designers of the Java language, would give any significant benefits.

7.4 Prototyping of Design for Registration and Interest Agents

In this section will be described work done towards establishing the feasibility of one part of the designs above, that concerned with Registration and Interest agents. These particular agent roles were chosen because they represented one of the smallest self-contained systems which could be built. Other agents, such as the Attention, Site Search, Mobility and Collation agents would rely on the existence of these, or make use of the information which they generate.

A prototyping methodology was used for this work and the first part of this section describes this methodology and its objectives.

7.4.1 Prototyping: Methodology and Objectives

The main features of a prototyping methodology have been described by Mazza et al (1994) as follows:

"The use of prototypes to test customer reaction and design ideas is common to many engineering disciplines. A software prototype implements selected aspects of proposed software so that tests, the most direct kind of verification, can be carried out.

Prototyping is the process of building prototypes. Prototyping within a single phase is a useful means of reducing the risk in a project through practical experience. The output of a prototyping exercise is the knowledge that is gained from implementing or using prototype software.

The objective of a prototype activity should be clearly stated before the process starts. Prototyping to define requirements is called 'exploratory' prototyping, whilst that for investigating the feasibility of proposed solutions is called 'experimental' prototyping.

Prototypes usually implement high risk functional, performance or user interface requirements and usually ignore quality, reliability, maintainability and safety requirements. Prototype software is therefore 'pre-operational' and should never be delivered as part of an operational system."

It can be seen from the above that the importance of a prototyping exercise lies not so much in the actual prototype system developed, but rather in the lessons learnt during its construction and the conclusions which can be drawn for application to future developments. The work here follows these guidelines, broadly falling into the ‘experimental’ category. The main objectives were:

a) to verify that the requirements and outline design given above for registration and interest agents are sufficient to construct a workable system;

b) to establish that no unforeseen problems arise in terms of, for example, stability or deadlock conditions;

c) to attempt to clarify the boundaries between these parts of the system which could be implemented directly in a generic form, and those parts which would have to be constructed as specialised components in any practical system;

d) to allow construction of an approximate performance model which can be used to predict the characteristics required of any practical system;

e) to evaluate the language and environment used for the prototype, both in terms of its suitability for this type of prototyping work, and in terms of its potential for delivery of practical systems;

f) using the results of this evaluation, to attempt to specify the characteristics which would be possessed by the ideal implementation environment for this type of work.

The characteristics desired in a prototyping environment need not necessarily be the same as those required of the environment in which the final implementation would take place. For prototyping, emphasis would be placed on:

a) speed of development;

b) ease of fault diagnosis;

c) clarity with which the structural features of the design can be expressed;

d) lack of any artificial constraints on the nature and structure of the design imposed by the prototyping environment;
Design and Prototyping Work on Generic Agent Roles

e) sufficient physical capacity to allow the design to be elaborated to the degree required by the prototyping exercise;

and rather less on the efficiency, portability and reliability needed in the final environment.

The WAVE language and interpreter environment (see chapters 2 and 5) was chosen for this work, largely because it possessed the degree of autonomous code mobility needed to implement the agents, and because the author was already familiar with its features. Being an interpreted language, its speed of development was also expected to be good. The main risk in this decision lay in the fact that this application would require considerably larger programs to be written in this language to implement the agent prototypes than had hitherto been attempted. Section 7.6 contains an assessment of how successful this proved to be.

7.4.2 Prototype Design

This prototype comprised four distinct types of agent whose collaborative behaviour determines the operation of the overall prototype. These are:

a) static nodal agents, one associated with each node of the processing network;

b) other static agents each representing a simulation model executing at one node;

c) a mobile registration agent for each simulation model;

d) mobile interest agents issued periodically by each registration agent.

The main functions of each agent were as follows, and are a sub-set of the functions listed in Table 7-1. More detailed descriptions, expressed in pseudo-code, can be found in Appendix F. Code for the agents, written in a commented variant of the WAVE language, will be found in Appendix G.

Nodal Agent:

- Publish position: Make the current position, time and vacancy level on this node available to registration agents;

- Load balancing: Obtain the number of registration agents currently using this node, and the positions and time stamps of their models. Do the same for the positions of the
Design and Prototyping Work on Generic Agent Roles

... immediately neighbouring nodes and perform a load-balancing operation by moving the centre position of own node.

**Simulator Agent:**
- Operate the equations of motion of the simulator entity;
- Launch a registration agent;
- Periodically receive a list of interested simulators from it;
- Reply with an update of own model's position and status;
- Transmit own position and status direct to all interested simulators.

**Registration Agent:**
- Navigate over processing network until positioned on the node closest to one's own model position;
- Publish existence of own model and its position at the current node;
- Periodically issue an interest agent to detect other entities;
- Assemble detections from other interest agents into an interest list;
- Send interest list back to own simulator;
- Receive an update of position and status from own simulator;
- Assemble detection reports from own interest agents into a list of entities which will transmit information to one's own entity.

**Interest Agent:**
- Spread outwards for a pre-determined distance;
- At each node, discover registration agents for other entities;
- If they were detected, notify them and own registration agent of this.

These various agents need to communicate using the facilities provided within the WAVE language. This provides only a limited set of facilities:
agents from the same source can communicate by sharing nodal variables, provided that they are on the same node of the WAVE graph;

communication between agents on different nodes is only possible by code mobility, i.e. a thread of control must migrate to the other node, message passing not being permitted;

agents from different sources can communicate only through the topology and labelling of the WAVE graph, i.e. by creating new nodes and links whose labels can subsequently be read by other agents.

Within the bounds of this restricted set of operations it should be possible to prototype any required communication scheme, though the extent to which this is a practical approach will depend on the application. Section 7.6 contains comments on the practicality in this instance. The communication scheme adopted for this prototype is described below.

As the Interest and Registration agents came from the same source, they could use nodal variables for communication. When the interest agent detected another entity, it spawned a new program thread which was sent back to the Registration agent's node, setting nodal variables there with the identity of the detected entity. These could then be read by the original registration agent's thread.

Similarly, as the registration and simulator agents came from the same source, they could also use nodal variables for communication. The registration agent periodically spawned a program thread which was sent back to the simulator's node to set a nodal variable there containing the interest list which had been built up. Whilst there, it collected information from other nodal variables which the simulator had set containing the entity's position and status, and returned to its original node to set these into yet more nodal variables which could be read by the original registration agent's thread.

The communication between the registration and interest agents from different simulators, which leads to discovery, had to take place through labelling of the WAVE graph, since these agents had originated from different sources. Similarly, any communication between these agents and the nodal agent took place through the graph. The scheme adopted is illustrated in Figure 7-1. It involved creation of new nodes, called here 'data nodes', with labels conveying the information,
which were joined to the existing nodes via directional links. The labels of these links indicated the type of their data node.

Each nodal agent set up a link labelled ‘+h’ (for ‘home’) to a data node labelled with its node’s position, current time and number of registration agents. It also set up another labelled ‘+n’ (for ‘neighbours’) to a data node containing the positions and occupancies of its nearest neighbours, together with the label of the link through which each neighbour could be reached. These data nodes contained the basic information used by registration agents for their navigation function.

As each registration agent reached a new node, it would create a data node containing its identity and position in space, reachable through a link labelled ‘+r’. These were accessed by the nodal
agent to perform its occupancy limit and load balancing functions, and by the interest agents to perform their discovery operations. When an interest agent had successfully both discovered and detected the registration agent of another entity, it would communicate the fact by adding its own temporary data node, with a link labelled '+i', to the data node belonging to that registration agent. The registration agent would then acknowledge receipt of the information by removing this temporary data node.

This communication scheme is capable of being extended to accommodate other types of information, for example, information being maintained by various environmental agents could be held on data nodes with links labelled '+e', while the presence of a simulator capable of running certain types of entity model would be described on data nodes labelled with '+s' links. These would be used by site search agents to locate suitable processing resources. Note that all the links to data nodes need to be directional to indicate unambiguously which end of the link is the data node, and which end is its point of attachment to the rest of the graph.

**Generic Properties of the Prototype**

Inspection of the functions being performed by these agents indicated that, at least within the limits of the prototype functions, some of these agents might be implemented directly in their generic forms, whereas others would need to be instantiated in specific ways to accommodate the functions needed by particular simulators and models.

The specific agents would clearly need to be the interest agents, which in any practical system would contain approximate models of sensing processes, and the simulator models themselves. The nodal agents and registration agents would seem to be good candidates for generic implementation, since the only information required for their operation is the identity of simulator entities and position of entities and nodes within the simulation arena.

In any practical scheme, the interest agents would need to be provided with information about the signature (i.e. visibility) of the entity being detected. If this were to be delivered by a specifically defined Attention agent, then the registration agents could be kept as generic agents within this scheme.

Looking now at the component properties of this scheme, the most natural replaceable software component would be an extended one comprising a simulation model and all of the
interest and attention agents which it could give rise to. The discovery operation would need to be defined as a protocol governing the attention/interest interaction for any particular sensing mechanism, e.g. visible, infra-red or radar. Note that this protocol would not constrain the contents of the sensing model in the interest agent, merely the information which was available to it from the attention agent.

This prototype was developed by the author to operate over a network of three Sun Sparc workstations at DERA. The operation of the prototype was monitored through two display screens, shown in Figures 7-2 and 7-3. The first shows the network of geographical servers as a fixed structure, including the communications links between them, and shows which registration agents are occupying the various nodes of this network. The second shows the virtual space of the simulation, with the motions of the various entities shown, and the movements of the centres of the areas of interest of each of the geographical servers, allowing the operation of the load management scheme to be observed.

Figure 7-2 Network Display Screen Produced by Interest Management Prototype
In its final form it operated reliably, if rather slowly. Performance measurements are shown in Appendix D. These were used in the following section to estimate parameters within a performance model, used to predict the scalability of the scheme.

The development of this prototype was not without difficulty. It appeared to tax the agent language used to its limits, both in terms of the then capacity of its interpreter, and in terms of the human programmer's ability to code the required functions without error. It would have been desirable to have been able to extend the prototype system to accommodate some more realistic agent functions, but this could not be accomplished within the existing limitations. These aspects will be discussed further in section 7.6.

7.5 Performance Model for Registration and Interest Agents

The purpose of this performance model is to elucidate the main factors governing the performance of the prototype system, and by so doing, allow predictions to be made about the characteristics which would be required of a more practical system. This latter step involves several areas of large uncertainty and so the final result must be treated with considerable caution. The major areas of uncertainty are:
Design and Prototyping Work on Generic Agent Roles

- the speed of processors and communications links;
- the efficiency of language and environment employed;
- the complexity of the functions called upon by specific simulators.

As a starting point, the performance equations for the mobile model scheme, derived in section 5.4.2, was used as the basis for modelling the performance of this prototype, with some additional parameters to account for the existence of several different types of agent. In this model there are five basic time parameters, namely:

- $t_a$: basic time to execute a nodal agent, including communications with its nearest neighbouring nodal agents (msec);
- $t_g$: basic time for registration agent execution, including time to communicate with its simulator model, and time to periodically move it (msec);
- $t_p$: time for a registration agent to process entity information from the interest agents (msec);
- $t_i$: time for interest agent to detect an entity (msec);
- $t_x$: time to transmit an interest agent across the network (msec);

As before, the total number of entity models is $n_m$ while the total number of processing nodes is $n_p$. The average number of candidates for the interest list constructed for each entity is $n_i$.

The formula for the processor cycle time for this system needs to be extended to cover all of the agent types involved:

$$\text{processor cycle time} = t_a + t_g + t_p + t_i + t_x$$

where:

$$\text{reg. agent time} = (\text{number of reg. agents per node}) \cdot (\text{time per reg. agent})$$

$$= \left( \frac{n_m}{n_p} \right) \cdot (t_g + t_p \cdot n_i)$$
Design and Prototyping Work on Generic Agent Roles

interest agent time  = (number of int. agents per node) \cdot (time per int. agent)

= (\sqrt{n_i} + \sqrt{(n_m/n_p)})^2 \cdot (t_x \cdot n_m/n_p)

since the total number of registration agents interested in this node is that contained in a circle of
radius proportional to (\sqrt{n_i} + \sqrt{(n_m/n_p)}), and each agent needs to inspect all of the (n_m/n_p)
registration agents currently on this node.

int. agent transmission time  = t_x \cdot (total no. of agents coming in and out)

= t_x \cdot (\sqrt{n_i} + \sqrt{(n_m/n_p)})^2

since all local agents are sent out and all interested remote ones must come in.

inter-agent comms time = t_y \cdot (no. of remote int. agents) \cdot (no. of local reg. agents)

+ t_y \cdot (no. of local int agents) \cdot (no. of remote reg. agents)

= 2 \cdot t_y \cdot (n_m/n_p) \cdot (n_i + 2 \cdot \sqrt{n_i} \cdot (\sqrt{n_m/n_p}-1))

from the formula derived in Section 5.4.2.

The overall processor update rate then becomes:

\[
( t_y + (n_m/n_p) \cdot (t_x + t_p \cdot n_i) + (\sqrt{n_i} + \sqrt{(n_m/n_p)})^2 \cdot (t_x + t_i \cdot n_m/n_p)
\]

\[
+ 2 \cdot t_y \cdot (n_m/n_p) \cdot (n_i + 2 \cdot \sqrt{n_i} \cdot (\sqrt{n_m/n_p}-1))^{-1}
\]

In practice, this formula may prove to be pessimistic, since it assumes that all of the entities of
interest are detected by the sensing algorithm in the interest agent. In practice, a certain
proportion of them will not be detected, and the data transmission times and registration agent
processing times will be reduced by this factor. However, this will not alter the general form of
the scaleability curves given by this formula, which represent the worst case.

The method adopted was to take measurements of the performance of the prototype which would
allow estimate to be made of the values of the performance model parameters applicable to it.
This was done by speed measurements covering a variety of conditions, single processor and
multi-processor and with various numbers of models. The results are shown in Appendix D. It
was not possible to obtain accurate estimates of all of the parameters in this way, but the
approximate estimates obtained are sufficient to achieve the main objective of this exercise, which is to explore the main scaling properties of the registration and interest agent system.

The predicted update rate achieved as a function of number of entities and number of processors is shown in Figure 7-4, in a form analogous to the performance plots for the mobile models system of Chapter 5. The basic data used to produce this and the next three plots is in Appendix E. That the system is indeed scaleable can be verified by observing the invariance of performance with simultaneous increases in both entity and processor counts for large overall systems sizes. It should be noted, however, that a 200 times improvement in performance over the benchmarked figures has been assumed in this figure in order to provide a reasonably practical update rate scale (see below).

As an aside, Figure 7-5 shows the speed-up factor due to the use of multiple processors. This exhibits a similar super-scalar effect to that of Figure 5-8 in Chapter 5, for example, using 1,000 processors gives a speed-up of 100,000 times due to the effects of geographical sorting in reducing the overall computing task to be performed.

![Figure 7-4](image)

*N.B. A 200 times execution performance increase has been assumed*

**Figure 7-4 Update Rate Prediction for Interest Management Prototype with 30 interested entities**
Figure 7-5 Speed-up due to Multiple Processing for the Interest Management Prototype

One other notable characteristic is the rapid fall-off in performance as the number of entities per processor, $n_m/n_p$, increases, which is accounted for by the square law variations of both interest agent time and inter-agent communications time with $n_m/n_p$. This effect justifies the emphasis placed on the need for load balancing in Section 7.2 above, to place limits on the maximum number of entities per processor, irrespective of the distribution of entities in virtual space.

Another aspect of performance is the variation of achievable update rate with the range of the sensor being simulated. In the performance model above, this is determined by the parameter $n_i$. The effect of increasing $n_i$ by a factor of ten is shown in Figure 7-6. As can be seen, although the performance does change, it does not do so by the same factor due to its strong dependence on $n_m/n_p$. This is an encouraging result, since it indicates that the penalty incurred for simulating very long-range sensing algorithms need not be in proportion to the amount of information obtained from them.
Design and Prototyping Work on Generic Agent Roles

Figure 7-6 Update Rate Prediction for Interest Management Prototype with 300 interested entities

Finally, it will probably have been observed that the absolute performance predicted falls short of what would be required of a practical system by a factor of around 200. To achieve a reasonable level of responsiveness in the context of a manned simulator, the discovery process should operate with an update rate of at least 0.5Hz. This has been demonstrated on the HOVERS simulator, where discovery operations are performed at a much lower rate than dynamics calculations, and rates of 2Hz have been found more than adequate for the representation of computer-generated ground forces and missile seekers.

As noted at the start of Section 7.5, the actual level of performance of a practical system would be strongly influenced by processor/network speed, language efficiency and complexity of algorithms employed. The tests above were performed on relatively slow 28MHz processors with a 10 Mbit/s Ethernet as network; one could safely assume a factor of ten to twenty improvement in speed for both. Against this, the complexity of the functions being performed could be expected to rise by at least a factor of perhaps 3 to 5 in a practical system, due to the need to simulate complex sensors, operating within a dynamically variable environment, and handling a variety of signature data. It would seem that the only way in which adequate performance could
be reached would be through significant improvements in language efficiency, by a factor of between 30 and 100 times.

7.6 Discussion & Critique

The work reported in this chapter has gone some way to establishing the basic feasibility of some of the generic agent roles outlined in Chapter 6, paying particular attention to the roles of Registration and Interest agents. Many of the significant design problems have been exposed, particularly the need to ensure good dynamic load balancing, and the need to incorporate features into the architecture to ensure robust operation in the face of failures of individual system components. The performance model developed has confirmed the basic scaleability of the system, and has given some indication that it could be suitable for modelling sensor interactions operating over a variety of scales.

The prototyping work described in Section 7.4 has confirmed that it is possible to construct very basic examples of registration and interest agents which can operate in a stable manner and respond to dynamic repositioning of entity models within the simulation arena, including the basis of a load balancing scheme. The work has indicated that it may well be possible to construct at least some of these agents directly in a generic form, with consequent savings in development effort and improved reliability following from standardisation of function.

However, this work stops some way short of being a full feasibility demonstration, since it proved impractical to introduce many of the features which would need to be present in a practical simulator. These would include detection algorithms, which need information about the external environment and the terrain through which the detection is occurring, as well as the signature determining the object's ease of detection. Improved load-balancing schemes providing more representative geometric limits on the motion of the server's centres of interest would also be of benefit. Finally, the cross-monitoring checks between agents, needed to enhance the robustness of the overall system, would have been desirable.

The main difficulties encountered followed from the acknowledged risk involved in the use of the WAVE mobile code language for much larger agent programs that it had been intended for, which gave rise to difficulties in coding even medium sized agent programs. There were two main aspects to this problem:
there was a lack of partitioning within a program and a single global name-space for all identifiers;

- the only construct provided for communication across the network is mobility of the thread of program control across the program graph, resulting in an overloading of the graph with many different types of information.

In applications where a single agent needs to perform several functions, the effect of these features was to make it difficult to compartmentalise the code for each, and to avoid mutual interference. Name-space conflicts could be resolved by adding prefixes relating to different sections of program, but at the expense of increasing the program size at run-time, and hence the time taken to transmit programs across the network. The conflicts introduced by the need to invoke program mobility for all network communications proved much more difficult to resolve.

For example, the registration agent must as a minimum maintain a two-way conversation with its entity model, navigate itself across the server network and receive communications from interest agents. In WAVE, all of these actions require a thread of control to move across the program graph, since much of the information needed for these functions is embedded in the graph as labels on various nodes. Where several functions are combined in one agent, it is essential to ensure that the movements made for one function do not conflict with the movements made for the others. There are three approaches to resolving this problem, none of them entirely satisfactory:

a) ‘Please leave the agent where you found it’. This involves ensuring the combination of moves made for any function are circular, and leave the agent back at its original location. One function, the navigation one in the example of the registration agent, is regarded as the outermost, and is absolved of the responsibility of making circular moves.

b) Splitting into separate threads, each of which performs the moves required for one function, and then communicates the results back to the place where they are needed.

c) Executing certain functions in conventional languages in a hybrid scheme.

In practice it is very difficult to guarantee that any combination of arbitrary moves will be circular, since the actual moves made result from a complex interaction between the ‘hop’ instructions in the agent code, and the actual topology and labelling of the graph, which is
Design and Prototyping Work on Generic Agent Roles

altering dynamically as a result of the actions of other agents. Subtle errors can be easily introduced, resulting in mispositioning, or in extreme cases, multiple or missing threads. Such errors are difficult to diagnose, since they may result from the asynchronous interaction of several agents.

The multiple threads solution is hardly less problematic, since, although some degree of functional partitioning can be achieved, new and completely artificial problems of communication and synchronisation between the different threads now comprising the agent are introduced. These have the effect of lengthening the programs and reducing the clarity of their structure and also have a pronounced effect on their execution efficiency.

The third solution, that of introducing separate functions written in conventional languages to supplement those written in WAVE, has several drawbacks. It also reduces program clarity due to the introduction of the artificial constructs needed to communicate between the sections of code written in different languages. Program efficiency suffers in cases where separate Unix processes need to be started to execute the hybrid functions. Flexibility is greatly reduced, due to the need to have functions pre-compiled and distributed ready for use, or linked into every copy of the interpreter.

While it is undoubtedly successful in implementing those types of relatively compact agent for which it was originally intended, it became clear by the end of 1995 that the WAVE language was not the ideal vehicle for the more complex types of agent needed for this work, and no further development was attempted using it after this date. However, the experience of constructing this prototype has allowed consideration to be given to the major features of a language environment which would influence its suitability. These include:

a) the programming model(s) supported,

b) application program interface (API) facilities available,

c) type and degree of modularity between sections of program,

d) execution efficiency,

e) portability,

f) quality of support for development,
Design and Prototyping Work on Generic Agent Roles

g) size of user community.

The first three characteristics mainly determine the difficulty of designing the program for any function, and of introducing new functions, while the others determine the ease with which performance targets can be met and reliable operation maintained over time and across platforms.

Based on the experience of this prototyping exercise, and the earlier work on MultiSIM and the HOVERS simulators, the list of characteristics for an ideal language for this purpose would include:

- code mobility as only one its supported constructs,
- conventional inter-process interfaces through shared files, pipes or memory areas,
- conventional inter-node communications through support for networking,
- good modularity, with the object-oriented paradigm preferred,
- support for multi-threading, with easy communication and synchronisation primitives,
- dynamic linking of classes to allow run-time combination of interchangeable components,
- high efficiency, close to that of compiled code,
- robust code transfer mechanisms to eliminate corrupted agents,
- wide-spread acceptance and availability on a variety of platforms.

It was at this stage, at the close of 1995, that a new language began to gain currency which had at least the potential to fulfil all of these requirements. That language was Java, which will be the subject of the final prototyping exercise, reported in the next chapter.
Chapter Eight
Chapter 8 Description of the Operator Interaction Agent Prototype

8.1 Introduction

New programming techniques involving mobility of executable code are being developed for various applications on the Internet. These range from the relatively simple transmission of object classes used in Java (Gosling & McGilton, 1995), intended initially to make world-wide-web pages more interactive, up to implementations of true mobile agents, able to relocate themselves autonomously over a network during execution, such as those found in Agent Tcl (Gray, 1995), Tacoma (Johansen et al, 1995), Telescript (White, 1994) and CyberAgent (FTP, 1996).

This chapter reports the results of a prototyping exercise in which an example of an agent fulfilling the Operator Interaction role (Chapter 6) was designed and implemented using some of these techniques, up to the point at which it could prove of use in the context of a real simulation. One of the more satisfying aspects of this work was that it presented the opportunity to re-establish facilities which were provided in a control design language called TSIM in 1976 (Winter, Corbin & Murphy, 1983), which operated on a single processor. Within TSIM, it was possible to monitor and, if applicable, alter all of the states and parameters of a model, and to produce on-line graphs of their time histories. This proved to be a very powerful facility for designing practical control systems, particularly when they incorporated non-linear elements, and hence could not be fully analysed by the linear analysis features of the language.

The agent work described here provides these facilities in the context of a distributed simulation, in a form which would work over a wide area network. It has been made much easier to use by providing a Graphical User Interface, which can incorporate specialised elements designed specifically for particular models. The application is intended to illustrate one way of alleviating some of the structural problems which have become apparent as distributed simulations have been extended in their scale and scope. These problems include:

- the difficulty of adequately monitoring and controlling models running at remote locations;
- the inflexibility introduced by the need to standardise all of the communications protocols used between the various components of a distributed simulation.
Relatively little attention appears to have been paid to these aspects in the literature. Any distributed system introduces the problem of remote monitoring, or diagnosing potential fault conditions within models not directly accessible to the operator. In a wide area simulation only limited information about the behaviour of remote models is usually available, such as their inputs from and outputs to other models within the simulation, and sometimes a standardised statement of their internal state, such as is available through the DIS Data PDU (IEEE, 1994a).

The limitation of this approach is that only certain data items can be accommodated within this pre-defined standard, and so it is not possible to fully monitor the internal behaviour of an arbitrary model in terms which the designer of that model would recognise as a full representation of it. Neither is it generally possible to control the model’s behaviour by making arbitrary changes to the parameters which influence its behaviour; for example, only a certain restricted list of parameters can be controlled through the corresponding DIS Send Data PDU.

This chapter describes an approach to solving the monitoring problem which is based on the use of mobile agents. Each agent is specialised to the monitoring of a particular class of models, and is potentially capable of displaying or controlling any of its internal parameters and states, but being mobile, can be made available for use at remote operator’s stations. This solution also illustrates one way of alleviating the second of the structural problems above, that of the need for an excessive number of agreed standards, whose limitations impose constraints which reduce the flexibility of distributed simulations.

One of the intentions of this work is to demonstrate that, at least for the monitoring function, no agreed protocol standards are strictly necessary. The agent solution described here makes use of specialised protocols between each model and its agent. These are determined by the site responsible for the model, and do not need to be agreed in advance with any other site, neither are they restricted to the contents of any pre-defined dictionary. Updates to the model and agent can be synchronised by this local site. The only agreed standard used is at the operator’s station, where the standard browsing mechanisms designed for use on the world-wide-web are used.

This problem is also being partially addressed elsewhere by proposed developments to the DIS standard based on the use of an extensive data dictionary to provide more flexibility within the protocols (Valentine, 1995). However, this solution would still involve the need to reach
agreement within the distributed simulation federation as to which elements from the dictionary would be used for any particular exercise.

The solution described below has been implemented as an example of an ‘ambassador system’ (see Section 2.3), since the interaction agent clearly represents the interests of the remote model at the local monitoring station, but there is no need for it to possess autonomy of motion, or to preserve its execution state after moving. Further analogies along these lines could be drawn, for example, the private communications link between model and agent is analogous to a diplomatic bag, especially in view of its undeclared content, and the protected execution environment of the agent could be likened to an embassy compound. While it is clearly possible to take these analogies too far (e.g. agent meeting place equivalent to a diplomatic cocktail party), their existence does lend credibility to the belief that useful real-world functions could be performed by such ambassador systems.

8.2 Description Of Prototyping Work

To fulfil its purpose of monitoring the internal operation of a variety of remote models, in terms which the designers of the models would recognise, requires that the software providing the interface be specialised for each model, with the appropriate displays and graphical user interfaces needed to adequately appreciate and control the internal states of each model. The main requirements for the computing environment for this application are:

- it should support mobile code, so that the monitoring agent can be maintained at the modeller’s site, and used at the operator’s.
- it should provide mobile GUI definitions, capable of operating on a variety of computing platforms without modification.
- it should for preference be object-oriented, to encourage ready re-definition and extension of existing software. The need for this will become apparent in section 8.2.1 when discussing how to derive specialised monitors from a generic template.
- speed of execution is not generally a problem in graphical user interfaces, so an interpreted language would be acceptable.
The language chosen to implement this prototype agent is Java, selected because of its mobile-code feature and the relative ease with which mobile GUIs can be implemented in it and run on any platform which supports the Java Virtual Machine standard (Gosling & McGilton, 1995). One convenient way of using Java is to define an “applet”, which is a partial application that can be accessed and run within a suitably equipped world-wide-web browser, under the control of a remote ‘html’ page. Although applets have some limitations necessitated by the need to protect local resources from untrusted remotely-defined code, they have proved perfectly adequate for the prototype described here.

In use, the selection of an agent would be guided by information contained in an html page detailing the simulation models available to be monitored at the remote site. After selection, the appropriate agent would be transmitted as a Java applet in byte-coded format to the operator’s browser. Once installed and running there, it sets up a private communications link to the simulation model over which the monitoring and controlling information is exchanged (Figure 8-1).

In the prototype described here, a standard TCP/IP link has been used between model and agent, with the simulation model as server, waiting for the agent, acting as client, to request a connection from it. If the model represents a class of objects, then the information for the various instances is multiplexed over a single connection. It should be noted that the TCP/IP standard is not essential for the operation of an agent of this sort - any protocol supported by the underlying network would suffice, since only the model and its agent need be in agreement on it.
8.2.1 Generic Agent Definition

This freedom to define new agents for each model could have the drawback of requiring too much work to be of practical use. It would not be desirable to have to start from scratch and write a completely new agent for each model.

For this reason, the approach adopted here is to attempt to define a generic agent which would be capable of monitoring a wide variety of models and would provide perhaps 80% of the functionality required of a fully specialised agent. The remaining capability needed would then be provided by extending the generic agent, using the inheritance facilities of the Java language, to produce whatever new facilities are required for any model, with considerable saving of effort in design, coding and debug.

The existence of this generic agent does not reduce the flexibility of the agent approach, since its use is entirely optional, and is merely an extension of the design choices available to the modelling team. It is still possible to start from scratch for any models for which the generic agent is not suited. Indeed, one could envisage eventually having available a selection of generic agents, each one applicable to a particular range of model types, from which the most suitable could be selected.

Neither would the evolving life-history of any particular generic agent design place any requirements on its users; when a new version of the generic agent is produced, users of older versions would not be forced to upgrade to it, since the design and upgrade status of each model/agent pairing is entirely independent of the others. The only dependency is on the Java computing environment and GUI API, and its interface to the browsing mechanism, which have now been well defined as a standard (Gosling & McGilton, 1995).

The generic agent described here is intended mainly for monitoring models representing classes of pseudo-continuous models, and gives access to the individual instances within each class. When loaded, it starts by opening an applet window with a control button inviting the user to activate the Class Monitor screen for that model.

Unlike the class browsers supplied with many object oriented development systems, which show the methods defined for a class before execution, this is a run-time monitor screen which shows a list of all the current instances of models running within this class, together with a colour-
Operator Interaction Agent

coded indication of the current status and behaviour of each one. This status is an enumeration which is user-defined for each class, and is intended to give the operator an immediate cue as to which instances are likely to be worth monitoring in the current state of the simulation (Figure 8-2).

Selecting one of these instances has the effect of creating an Instance Agent within the operator's browser, which opens up a window via which the operator can select sets of states and parameters to be monitored. This includes an annunciator area, on which significant discrete events and error message can be posted, controls for the communications link with the model, a repeat of the status display from the class browser window and a set of selectable display pages each of which displays the value of a different set of the model's states and parameters. The communications with the model are controlled on a per-page basis, to minimise the impact on the real-time performance of the network.

![Instance Agent](image)

Figure 8-2 Monitoring Window for a Complete Model Class

A graph plotting facility is provided to allow several parameters to be plotted as a function of time within another window, to monitor the dynamic behaviour of the model instance. Each parameter has its own area on the display page, showing the current value, a text entry field for modifying this value, where applicable, and a selector button for the graphing facility. Currently the parameter types accommodated include floats, integers, enumeration values and mutually exclusive sets of keywords. This list of types is implemented through an inheritance tree, and can readily be extended by introducing fresh classes within that tree.

There are two ways in which this generic agent can be adapted for use by any particular model. The most straightforward way is to configure the page displays on the generic instance agent for the particular parameters of that model. This is done by means of a configuration file in which the names and natures of all of the model's parameters and states are defined, together with the
status enumeration and its colour coding. The second method is more flexible and involves extending the functions available from the generic agent, defining new classes by means of inheritance, to produce a new and specialised agent which can have additional GUI windows.

The most likely extensions needed to specialise this agent for use with individual models would be to provide instrument-like displays for particular parameters, giving a more ergonomic monitoring interface, and to provide mouse-driven controls for specific parameters, e.g. steering inputs to a vehicle (Figure 8-3). Note that these controls would be intended for medium-bandwidth use only, and could not substitute for the high-bandwidth, low latency controls and displays needed for demanding piloting tasks.

Apart from the applet itself, this generic agent consists of a number of Java classes:

- ClassAgent provides the class browser GUI frame, and distributes the communications from the simulation model to the various instance agents to which it applies.

- InstanceButton displays the instance selection buttons within the class browser and is responsible for creating instance agents when selected. It also maintains the status display in both class and instance windows.

- InstanceAgent provides the main GUI frame for interfacing to a single instance, with the functions described above.

- AbstractItem is an abstract class which can be extended to display and optionally alter a specific type of state variable or parameter. Extensions for floats, integers and enumerations, all identified by a keyword, have been provided already, as well as for enumerations comprising sets of mutually exclusive keywords with no accompanying value.

Appendix H gives the interface specifications for these classes, in the standard Java format produced by the "javadoc" toolkit.

The prototype generic monitoring agent has been applied to the HOVERS distributed helicopter mission simulator, based on MultiSim and described in Chapter 4. It has been used to monitor the behaviour of certain of the ground forces models running on remote processing nodes. The simulator had been written in Ada, and the extensions to the models needed to allow them to
exchange information with their agents were prepared as an Ada/C hybrid. However no assumptions have been made in the generic agent’s design concerning the nature of the simulator model, and it is hoped that it could be applied to simulations with other internal structures with relative ease.

Specific extensions to the generic agent were also constructed, so as to make a specialised tracked vehicle monitor. This had an additional control/display window (also shown in figure 8-3) which contained the additional instrumentation needed for on-line interpretation of the vehicle’s situation. This comprised two strip-type displays, a linear one for speed and a moving-scale one for heading. Control of both speed demand and rate of turn was provided through a mouse-operated joystick, which could be moved in two dimensions over a patch of screen, together with control buttons to enable and disable it.

![Generic Graphing Window](image)

**Figure 8-3 Example of Monitoring Windows for one Instance of a Model Class**

The intention of this additional control panel was to allow the vehicle to be steered over the terrain model. In practice, it would be used in conjunction with the standard HOVERS “godseye view” screen, which gives a window into a fully-textured representation of the terrain, viewed from the vehicle, or any other selectable point. While the vehicle is being steered, the other monitoring functions of the generic monitor are still available to inspect the vehicle’s status, events and states, and alter them where necessary.
To give some indication of the amount of effort required to produce such specialised monitors, this relatively unsophisticated, though useful, display device was completed in one day's work by extending the classes of the generic agent, including the modifications needed to the Ada model to accept the additional control inputs. The generic monitor on which it was based needed several weeks to bring to completion.

Experience with the use of this monitor during simulation runs has not shown any significant degradation in simulator performance, provided that care is exercised over the amount of data being requested simultaneously from any one processing node. As the operator is in control of the number of instances being monitored, and the number of pages of information about each one being updated, this should not represent a significant practical limitation on the use of the technique.

Appendix I shows the format of the data exchange used between model class and agent for this design of generic agent. Text messages have been used for flexibility and ease of interpretation. Appendix J shows the format for the configuration file needed to inform the generic agent about the nature of the states, parameters and status indications to be expected from any particular model.

8.2.2 Experience with Java

Experience of using the Java language on this project has been very encouraging. The language itself has proved to be easy to work with, the strong support for the object-oriented paradigm allowing complex systems to be constructed with a minimum of redundant work, provided that sufficient thought is given to designing the best object structures beforehand.

The safety features built in to the language and its run-time system are impressive, and exceed the already stringent characteristics of Ada in a number of respects. In particular the trapping of uninitialised variables, the declaration of the need for exception handlers and the run-time checks on the validity of casting operations have all proved their worth in this project. Their presence greatly increases the confidence with which large software systems can be assembled and expected to function without major errors in program consistency. Belief in the soundness of the software engineering underlying Java was one of the factors which influenced the author in deciding to apply it to real-time distributed simulation.
One of Java’s best features lies in the range of standard APIs available for different purposes. The main ones used for this project are the Networking API and Abstract Windowing Toolkit, both of which proved to be comprehensive and exceptionally easy to make use of. As an illustration, producing a prototype client-server TCP link in Java using the networking API took 1.5 hours, starting with zero knowledge, while reproducing the server function in C for Unix took several further days of effort.

The portability and widespread use of Java mean that a number of libraries of generally useful and proven programs are being made available for use via the Internet. The graphing package used in the generic monitoring agent was adapted from one of these, with considerable saving in development effort. The main drawback to Java at present concerns the execution speed of the byte-code interpreters currently in use. To relieve this problem, many platform suppliers are currently bringing out “just in time” compilers which promise to improve execution efficiency by at least an order of magnitude for most applications.

Further developments to Java include the ability to define “servlets”, which are analogous to applets, but act in the opposite sense, allowing a client to send a piece of code to an accommodating remote server for itself and others to make use of. One of its main applications would be to provide access to remote data-bases, but could equally well be used for giving remote models control over the services available in distributed simulations. Further proprietary extensions to provide a fully autonomous mobile agent system, which may be capable of implementing some of the other generic roles described in Chapter 6, are becoming available, but have not yet been evaluated (FTP, 1996; Lange & Oshima, 1997).

8.3 Conclusions And Further Work

This chapter has described a mechanism for monitoring and influencing remote simulation models of arbitrary nature by making use of specialised mobile agents. One of the main purposes of this work has been to illustrate how a complete distributed system, with remote access, can be constructed without the need for defining any restrictive communications standards, resulting in increased flexibility and a restoration of local design autonomy, indeed, imbuing the simulation system with many of the desirable characteristics of the world-wide-web itself.
In the example application described above, this approach encourages monitoring agents to be produced by local sites to accommodate the needs of any variety of model for which they are responsible, and allows them to be made immediately available for use over the wide area network to give remote operators access to the complete range of information governing the behaviour of the distributed simulation. The prototype was implemented as an ambassador system, following the definition coined in chapter 2, since it needed only limited autonomy.

In future it is hoped to extend this principle to other aspects of the control of distributed simulations, including model distribution and execution start, scenario generation and set-up, monitoring and logging of communications, through the same standard browser interface as is used to monitor models.
Chapter Nine
Chapter 9 Conclusions

This thesis has presented various approaches towards the synthesis of several ideas which have recently been gaining currency in computing, namely: distributed real-time simulation, component architectures and mobile agent systems. Possible approaches have been presented as to how these techniques could beneficially be combined to enhance the capability and flexibility of computer simulations, and a number of prototyping exercises have been undertaken to establish the feasibility of some of the approaches suggested. The results have been published in a series of conference and journal papers (see page iv).

One application of component architectures to simulation was described in Chapter 3, where the design of the MulTiSIM distributed simulation framework was presented. This framework employs a component-based architecture, in which re-use of models is encouraged by ensuring that the models do not interact directly, but only through generic communication facilities provided by the framework. The models can be assembled into hierarchies representing the structure of complex simulation entities. This lends itself to multi-scale simulations, in which some entities of particular interest can be represented down to fine levels of detail, while others may be present only in a rudimentary form. This design was implemented by extending a conventional language, Ada83, by adding support for object oriented constructs.

The various components of these hierarchies can be distributed in an arbitrary way between the nodes of a local area network in order to satisfy the needs of load balancing or local information accesses. The operation of this system is supported by a distributed run-time data-base optimised for the high volume of accesses demanded by real-time operation of a simulation with many sensors, typical of a battlefield mission simulation. The generic data access mechanisms are influenced by the hierarchical structure so as to give a high degree of structural transparency to the inter-actions between entities, as well as providing for transparency over model type and distribution.

This is believed to be the first example of a system of re-useable components combining distribution and real-time operation with a transparent hierarchical structure. The system has been in use for three years for trials on the HOVERS mission simulator in DERA, during which time the original set of models have been re-used in many different configurations, and augmented by many other models. The original generic instantiations of communications
Conclusions

handlers, defined for the first trial, have also been extended incrementally to accommodate the needs of these new models.

The dynamic load-smoothing algorithm developed for the framework’s scheduler has proved of considerable benefit in presenting an illusion of the smooth flow of time to the human operators of the simulator, while running a mixture of models with widely varying update rates.

The framework above was designed for medium-scale simulation, comprising a few hundred entities; under these conditions it is practicable to broadcast information globally to all simulation nodes, a mechanism which unfortunately does not scale to larger simulations. It is interesting to note that a somewhat similar object-based approach is being taken within the new High Level Architecture for simulation, currently under definition for DMSO. Here, however, the intention is to support very large-scale simulations, requiring a high degree of selectivity in communications if benign scaling properties are to be achieved. The HLA definition contains publish-and-subscribe facilities, mediated by a Run-Time Infrastructure (McGarry et al, 1995), to aid in minimising the need for dissemination of information.

An alternative approach to large-scale simulation was the subject of chapters 5-7 of this thesis. This is more of a self-organising approach, requiring no central mediating elements, and based on the use of mobile agent techniques to perform a geographical sorting of entities, from which the communications paths required can be derived by local operations. Chapter 5 presented the results of a collaborative pilot study, which is believed to be the first application of mobile code techniques to simulation. In this work, all of the entities in the simulation were mobile, transmitted between processors by mobile agents, and communication was limited to the other entities currently on the same processor.

Chapter 6 developed these ideas further to accommodate the more usual cases where the majority of models would not be mobile, and where communication is required over a variety of ranges of the simulation arena. A number of novel generic roles for mobile agents were identified, whose interactions would provide the emergent functionality of the system. These roles included sets of registration, attention and interest agents which together could perform the communications management functions and site search and mobility agents which would be responsible for satisfying the computing resource requirements of those entity models which could be rendered mobile. The idea of specialised interaction agents, to mediate the interactions of closely-coupled pairs of entity models, or entities with human operators, was also introduced.
In chapter 7, the designs of some of these generic roles were elaborated further to consider some of the practical constraints under which they would have to operate, including load balancing, stability and robustness requirements. A scaling model was derived for the communications management functions, to illustrate how performance would be influenced by simulation scale and processor provision. Finally, a prototype system of mobile registration and interest agents was described, from the performance of which it was possible to make estimates of the characteristics which would need to be possessed by any practical implementation, and to recommend features of agent languages and support systems which were likely to be of most benefit for this type of application.

While some of the roles in chapter 6 could probably be implemented directly in their generic form, others would be specialised for use with particular simulation models or protocols. This gave rise to the suggestion that the owning model and all instances of its specialised agents would together comprise a replaceable component within the overall simulation. Chapter 8 presented an example of a system designed on this principle, when considering one of the interaction agent roles, the operator interaction agent for monitoring the behaviour of a single model. Here, the model being monitored, the agent performing the monitoring function and the communications protocol between them, since they originate from a single node, could be treated as a single unit for purposes of software maintenance.

The idea of an ‘Ambassador system’ was also introduced, as being one needing only a sub-set of the usual facilities of a full system of autonomous mobile agents. An ambassador moves only at the request of its owner or its receiver, need not preserve execution state after a move, and represents the interests of its owning system at the site of its receiving system through a private communications link. The operator interaction agent above proved to be an ideal candidate for implementation as an ambassador, and was prototyped using the Java applet standard API, in what is thought to be the first use of Java in conjunction with a real-time distributed mission simulator. The choice of language in this case was influenced by the high level of software engineering evident in the safety features of Java.

9.1 Possible further work

There are several possible ways in which the work described in this thesis could progress.

The simulation framework in Chapter 3 would benefit from a re-implementation in Ada95, which has a form of single inheritance, which could be called on to make much of the dynamic
Conclusions

binding emulation currently used by the framework to control models unnecessary. Note that it would not be possible to convert all of the current uses of dynamic binding in this way. The calls made on the models by the event handlers rely on a form of multiple inheritance, more accurately described as multiple interfaces, which is not directly supported by Ada95. It should, however, be possible to replace the control of the event handlers and data stores themselves by the framework, since these packages are derived by instantiating generic packages, which is a form of sub-classing over which the Ada95 dynamic dispatching mechanisms will operate.

The facilities of the event handlers could usefully be extended to allow broadcasting of events to a number of interested models simultaneously. This would be of great benefit when modelling discrete visible events, such as explosions, whose effects need to be shown from a number of graphics windows. It would also increase the level of compatibility with DIS, which employs a similar broadcasting mechanism.

Another area where improvements could usefully be made is in formalisation of the methodology used to design sets of models and their communications mechanisms for use within the framework. There is a clear need when defining complex sets of models, which may be assembled into a number of configurations, to ensure that the data expected by a model is in fact being generated in a valid form elsewhere. This need is particularly pressing where several data-stores and event-handlers are being called into play, and data is being generated at several levels within the component hierarchy.

One very promising approach described in Chapter 4 is to characterise the interaction between models in terms of protocols at a higher level that of the basic event handlers and stores. Each protocol comprises the totality of the data transfer between models for a particular purpose. In general, this is bi-directional, though asymmetric, and involves signals passing via several data stores and event handlers. The description of a protocol could include time-dependencies, for example state transition diagrams for the models involved, showing what possible sequences of signals would be valid within a protocol. These states and sequences, or in simple cases the entire protocol, would carry semantic labellings expressing the intentions of the designer, and allowing these to be verified against the corresponding parts of the models making use of that protocol.

The question then arises as to whether, and to what extent, it might be possible to automate the verification of models, and check the correctness of their integration within a given
Conclusions

configuration. It would also be very useful to automate the production of the various files needed to create that configuration, and which initialise models to the start-point of a particular scenario. Investigations are under way to assess the suitability of existing tools for this, starting with the DOORS requirements analysis tool (QSS, 1996), capable of maintaining a hierarchical data-base covering the requirements for a particular model, and describing its structure and relationship with others.

As a final point before leaving the MulTiSIM framework, it would be very interesting to assess the suitability of Java as a replacement for Ada, particularly in view of the close match between the “single inheritance plus multiple interfaces” models of object-orientation used by both Java and the framework, and encouraged by the superior run-time program correctness checks of the Java language. Some potential problems with Java for this application would be its lack of generics, currently used to define new data-stores and event handlers, and its lack of consistent real-time performance, largely due to the use of interpretation and garbage collection. Real-time versions of Java can be expected to become available in the near future.

Extensions to the agent work reported in Chapter 7 could be usefully undertaken, to prototype more of the generic roles described in Chapter 6, and take them closer to the functions which would be required in a real simulation, including interfacing to the various environmental models which would be needed. Investigation of the suitability of various mobile agent implementation systems, such as Aglets (Lange & Oshima, 1997), would be the first step in this.

Further extensions could be made to the ambassador work reported in Chapter 8 in order to provide a complete control and monitoring facility for a distributed simulation. This would include ambassadors for the object data-base as well as other components of the framework, allowing the provision of improved browsing mechanisms, which could encompass the component hierarchy as well as the class/instance structure implemented in the current class-based browser. Mechanisms could also be provided to integrate the launching of executables at the start of a simulation run through the same browser interface, replacing the platform-specific mechanisms currently used. All of these extensions would require the use of html files containing information about the configuration of a simulation and its models. One very promising approach would be to extend the use of DOORS to automatically generate these files, and also the model-specific ones needed to configure the generic monitoring agents, directly from its data-base.
References
References


References


IEEE (1994c) “Standard for distributed interactive simulation - Fidelity Description Requirements”, Institute for Simulation & Training, Orlando, FL.


References


Appendices
Appendix A  WAVE - Introduction and Brief Guide to the Language

In this Appendix a distributed programming paradigm and language called WAVE (Sapaty 1988) is described, which allows flexible implementations of dynamic distributed systems. The language acts in a spatial mode, with separate threads of control navigating through a labelled program graph.

Each computing node in a distributed knowledge network has a separate copy of an interpreter for WAVE (Borst 1992), which operates on threads of code as they are sent to it, passing any not intended for local processing to their destination nodes. These threads can control their distribution over the network, splitting into concurrent branches when necessary, while operating on knowledge contained in the network, encoded in both its topological structure, and in arbitrary labelling which can be attached to both nodes and links.

The rest of this section has been written in the style of a brief tutorial. The code fragments given are intended merely to illustrate particular language features, and do not necessarily represent the most effective way of performing their function.

![Figure A-1 Example of a simple WAVE program graph](image)

WAVE programs operate over graphs in which nodes and links (arcs) have arbitrary labels, as in Figure A-1. A link may have a direction associated with it, shown by a ‘+’ sign preceding the label. A WAVE program (or wave) is divided into elementary moves and starts executing on a single node of the graph. It can contain moves which can cause it to jump to one or more other nodes, taking the rest of the program and any relevant variables with it. For example, a wave applied at node ‘a’ would jump to node ‘b’ over link ‘2’ by executing the move 2#b. Putting a sign on the link label enables jumping only in the appropriate direction, e.g. +2#b would only
allow jumping in the direction of the link, while -2\#a would only allow jumping in the opposite direction.

If the link label is omitted, then any appropriate link will be used, as in \#b, and omitting the node label causes jumping to all appropriate neighbours regardless of their labels. In this example a simple \# would result in the program being broadcast to ‘b’ and ‘c’ nodes, and continuing its execution on both as two parallel branches. It is possible to make jumps to nodes in the graph where no intervening link exists. These are referred to as tunnel jumps, for example @\#a would jump to all nodes with label ‘a’ from anywhere in the graph.

Each program takes with it independent copies of its travelling, or frontal, variables. Nodal variables stay behind on a node when the wave moves on, and provide one of the mechanisms by which separate parts of a wave program can communicate after they have split into parallel branches. Variables in WAVE are untyped and can contain integers, strings or vectors, which are constructs that can be used to emulate arrays, lists, queues, stacks, etc., and to perform many string manipulation functions. The language may be readily extended to accommodate further data types as the need arises. Frontal variable names start with ‘F’, while nodal variables start with ‘N’.

Certain special variables are pre-defined:

T assignment to T causes writing to a Terminal node.

C Content (label) of current node (need not be unique).

A Address of current node (unique to each node).

P address of Previous node for this program.

L Link label for the last link traversed

S Sign (+/-) of last link, if directional.

Moves which modify variables take two arguments, and modify the first one, e.g.

Ft=27 sets the frontal variable Ft to 27.
\( \text{Ft} + 1 \) increments Ft.

\( \text{N2} - \text{Ft} \) subtracts Ft from the nodal variable N2, leaving the result in N2.

\( \text{Fv} = \text{alpha} \) sets Fv to a string value ‘alpha’.

\( \text{Fv} = \text{alpha}; \text{beta}; \text{gamma} \) sets Fv to a vector of three strings.

\( \text{Fv} \& \text{delta} \) adds another string to the end of the vector.

\( \text{Fv} \% ' ' \) converts a vector into a string, replacing the delimiter with spaces.

\( \text{Fv} | ' ' \) splits a string into a vector at every space character.

\( \text{Fv} : 3 \) replaces the vector with its third element, (gamma).

\( \text{Fv} :: \text{beta} \) sets Fv to the index number of the element containing ‘beta’, (2).

Moves which are to be executed in sequence are separated by dots, e.g.

\( \text{Ft} = 27 . \text{N2} = 0 . \text{N2} - \text{Ft} . \text{N2} + 1 . \text{T} = \text{N2} \) results in -26 being printed.

Moves which are to be executed in parallel are separated by commas, e.g.

\( (\text{Ft} = 27, \text{Ft} = 28, \text{Ft} = 29) . \text{T} = \text{Ft} \) results in three branches of program, each with a different value of Ft, which will each continue on in parallel to execute the moves following the brackets, in this case to print Ft as: 29 27 28 (in arbitrary order).

Logical filters can be applied, which if true will allow a program to continue, and if false will cause it to terminate e.g.

\( (\text{Ft} = 27, \text{Ft} = 28, \text{Ft} = 29). \text{Ft} < 28 . \text{T} = \text{Ft} \) would prune the three parallel branches back down to one again, printing 27. The filter operations allowed are: == (equal to), /= (not equal), <, <=, >, >=, ~ (member of vector), /~ (not member). A branch can also be forced to terminate by the move ‘!’. Failed jumps also act as filters, i.e. if an impossible jump such as \(-2\#c\) is specified, that branch of program will terminate. This property is very useful when writing programs which spread
branches over an existing graph searching for patterns of nodes and links - any combination of
links which does not fit the pattern will cause one branch to terminate, leaving only the branches
which match the pattern.

The final major feature of WAVE is the ability to apply control constructs (rules) to parts of a
program. These can be used to create or modify the graph structures or to synchronise the
development of different branches of a program.

The **CReate** rule enables the creation of nodes and links if they do not already exist, e.g. the
graph of Fig. A-1 can be created by the program:

\[
\text{CR}(\#a, \text{Fa}=A, '2'\#b, '4'\#c, '-3'\#Fa)
\]

note how the absolute address of
node ‘a’ has been recorded in frontal variable Fa, so that a return can be made to the same
node later.

The **RePeat** rule causes the enclosed wave to be repeated until it terminates before the program
continues any further, e.g.

\[
\text{FV}=a, b, c, d, e, f, g.
\text{RP}(\text{Fi}+1, \text{FV}=\text{FV}, \text{Fi}+1, \text{FV}, \text{Fi}+1, \text{FV})
\]

results in the
printing of each element of the vector FV in turn until an empty one is found. The index
variable Fi is initialised as zero (or empty) automatically. A better program would be:

\[
\text{FV}=\text{FV}, \text{FV}^\prime, \text{T=FV}.
\]

\[
\text{RP}(\#, \text{N1}>0, \text{T=N1}, \text{N1}=)\cdot \text{T=C}
\]

will spread over a graph, printing any non-zero values
of nodal variable N1 that it finds, and setting them to zero. Eventually all branches of the
program will encounter zeros, exit from the loop and print their current locations.

The **SeQuence** rule forces parallel branches to execute sequentially, i.e.

\[
\text{SQ}('\text{wave1}', '\text{wave2}', \ldots)
\]

will ensure that ‘wave2’ will not execute
until all the branches originating in ‘wave1’ have terminated, however far over the graph
they may have spread.

The **Or Sequential** rule will try a number of possible alternative waves in turn until one of them
executes successfully, when it continues with the rest of the program, i.e.
OS( ('wavel'), ('wave2'), ('common tail') ). 'rest' will first execute 'wavel' followed by the common tail. If this succeeds, no other alternatives are tried and 'rest' is executed. If it fails, then wave2 and the common tail are tried and so on, until all alternatives are exhausted.

OS( #. 'common tail' ). 'rest' will try in turn each of the alternative destinations specified by the jump, together with the common tail, until one succeeds, e.g.

OS (:@#.C==b). T=A will jump to each node in turn, test to see if it has label 'b', and print the absolute address of the first one found, whereas @#.C==b.T=A would print addresses for all nodes with label 'b'.

Other rules include AndSequential, AndParallel, OrParallel, InDivisible and Wait. The synchronising rules all rely on the maintenance of a track record, through which the ultimate success or failure of an individual wave can be echoed back to the point at which the wave originated, and used to conditionally activate further waves.

Tracks and Nodal Variables are two mechanisms through which different branches of a wave program can interact. The third mechanism is through the dynamic modification of the graph structure itself over which all the waves are navigating.

WAVE programs are currently executed by an interpreter written in C under Unix (Borst 1992). When used on a network of workstations, each processor is initialised with a separate copy of the interpreter, which takes responsibility for a sub-set of the nodes making up the problem graph. The interpreters then send partial WAVE programs and frontal variables to each other as the program evolves. The extremely concise syntax of the language results in part from the requirement to be able to send programs over medium-bandwidth data links efficiently.

As an example of a complete program, consider the following to find the length of the shortest path tree in a graph, starting from node 'a':

@@a.SQ (RP (#.F+L.F<N,N==N=F),RP (#.N>0.N&C.T=N,N=))

In the graph, the links are assumed to have been labelled with their lengths. The program splits into multiple copies as it passes over the links in the graph. Each copy maintains a running total
of the distance it has travelled by adding the link length, $L$, onto a frontal variable $F$. This is then compared with a nodal variable $N$, which holds the minimum length by which any wave has so far reached that node. If this wave has a new minimum path length, or is the first to reach this node, it copies $F$ into $N$ and continues on to another node, otherwise it dies on the spot. When all these initial waves have died out, a second wave is sent over the graph to print the minimum path length to each node, together with its content label, $C$.

The WAVE interpreter is able to extend its functionality by starting Unix processes, and passing parameters to them from some branch of a WAVE program. The results are passed back to the initiating branch of the program, which suspends itself during the execution of the Unix process, e.g. `5?sleep` will suspend a branch for five seconds.

Since wave programs are interpreted, it is possible to modify them during execution. This is done by constructing the code to be executed as a string held in a variable, using the string handling facilities of the language. Quoting the name of the variable as a move causes execution of the code. Using this technique, recursive procedures can readily be implemented.
Appendix B  APIs for Interactions of Models and MulTiSIM Framework

B.1  Model Control Operations

The following operations are defined within each modelling package, and are called by the MulTiSIM framework to control the models. Starred operations are not applicable to single object models.

The following must be present in all classes of model:

- **Create_Class**: Defines the class object within the database.
- **Create_Instance**: Defines one instance of the class within the database.
- **Destroy**: Removes an instance of the class from the database.
- **Initialise_Instance**: Set up instance attributes before a run.
- **Checkpoint_Instance**: Write all attributes of an instance to disk.
- **Restore_Instance**: Read back all attributes of an instance from disk.

These must be present in all continuous-time models:

- **Update_Interval**: Returns the time update required by all instances of a class.
- **Collect_Inputs**: Retrieve any inputs required from data-stores.
- **Integrate**: Perform integration up to some time value.
- **Distribute_Outputs**: Send outputs generated to own data-stores.

Optional Operations:

- **Alter_Instance**: Send a command string to change instance attributes.
- **Alter_Class**: Send a command string to change class-global attributes.
- **Alter_Update**: Change the update interval of all instances of a class.
- **Checkpoint_Class**: Store class-global attributes to disk.
- **Restore_Class**: Read back class-global attributes from disk.
- **Clone**: Copy attributes and components of one instance from another.
Display_Instance  Display current attributes of an instance.
Display_Class    Display current class-global attributes.
Initialise_Class Set up class-global attributes before a run.
End_of_Run       Perform a clean-up at the end of one simulation run.
End_of_Simulation Perform a clean-up at the end of all simulation runs.
Execute          General-purpose discrete-event entry point.
Start             General-purpose entry point for starting a discrete-event model.
Stop              General-purpose entry point for stopping a discrete-event model.

In addition, the model designer may define any number of specialised operations to be called by
discrete-event handlers. These can transmit parameters in addition to the model identity.

B.2 Operations available for Putting and Retrieving Data within Data Stores

Data stores are implemented as Ada generic packages. Each instantiation can hold one type of
data record per object (its “data_item” parameter), and will define the following operations for
use by models.

Put_Local_Data
   This procedure is used to place data into the data store

Get_My_Data
   This procedure retrieves data belonging to an object from this data store. The time
   returned is the time at which the data was last updated. If this store does not contain data
   belonging to this object, then a No_Data exception will be raised.

Get_Any_Data
   This procedure is similar to Get_My_Data, except that if the object does not have any
   data in this store, then a search will be done through all the owners of the object for one
   which does have data. This Data_Owner will be (optionally) returned. If no owners have
   data in the store, then again No_Data is raised.

Remove_Data

xx
This procedure removes the object's data from this store.

Get_Total_Sources

This procedure sends back an array containing the references to all objects which have data in this store.

Source_Count

This function returns the total number of entries in the data store.

Has_Own_Data

This function returns True if the object has data in this store.

Has_Any_Data

This function returns True if the object or any of its owners have data in this store.

Data_Owner

This function returns the owner of the data send back by Get_Any_Data, i.e. either the object or one of its owners, or Meta_Obj if none have data in this store.

Active_Parts

This function returns a list of any parts of an object (including itself) which have data in the store. Note that this is necessarily a slow routine, since a tree search is involved.

B.3 Facilities Available from an Event Handler

Event handlers are also implemented as Ada generic packages. Each instantiation will handle a set of related events, with parameters. The events and parameters must be defined as a variant record type, which is the "local_event" parameter of the generic instantiation. Each event handler defines the following operations for use by models.

Register

This procedure informs the event handler that an object is able to receive events from it. If an event is queued for an object which has not registered, then a search is made up through all its owners to find an object which has registered. The event is then queued for this instead, allowing events to be handled at the most appropriate level in a hierarchy of parts. If no registered object is found, then No_Handler exception is raised.
Deregister

This procedure removes the registration for the object with this handler.

Schedule_Relative

Schedule_Absolute

These procedures place an event for an object on the queue belonging to this handler. Schedule_Relative performs the event at a time in the future relative to the current simulation time (now, by default). Schedule_Absolute performs the event at a time relative to the start of the simulation. The source of the event can be given to indicate the identity of the object issuing the event.

Cancel_Next

This procedure will cancel the next event of a certain type for the object within this handler's queue. Determining when an event is sufficiently similar to the event specified to be cancelled is done by the Equivalent function, supplied as a generic parameter when defining the handler.

Cancel_All

This procedure cancels all pending events for an object in this handler's queue.

Can_Handle

This function returns True if an object itself is registered with this handler.

Has_Handler_For

This function will return True if an object, or any of its owners, is registered with this handler.

Handler_For

This function returns the reference to the object which will actually respond when this handler receives an event for an object, i.e. it is either the object, or one of its owners, or Meta_Obj if none are registered.
## Appendix C Performance Predictions for Global Broadcasting and Mobile Model Systems

### Update Rate (Hz) predicted for a system with global data broadcasting

<table>
<thead>
<tr>
<th>No. of Models</th>
<th>No. of Processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>8192</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2048</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1028</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>512</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>12</td>
<td>18</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>14</td>
<td>23</td>
<td>34</td>
<td>45</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>13</td>
<td>24</td>
<td>40</td>
<td>60</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>12</td>
<td>22</td>
<td>39</td>
<td>65</td>
<td>97</td>
<td>129</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>17</td>
<td>32</td>
<td>67</td>
<td>94</td>
<td>141</td>
<td>187</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>41</td>
<td>74</td>
<td>122</td>
<td>162</td>
<td>241</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>86</td>
<td>143</td>
<td>213</td>
<td>282</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Update Rate (Hz) predicted for a system with global data broadcasting and data-bus saturation

<table>
<thead>
<tr>
<th>No. of Models</th>
<th>No. of Processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>8192</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2048</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1028</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>512</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>12</td>
<td>15</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>14</td>
<td>23</td>
<td>31</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>13</td>
<td>24</td>
<td>40</td>
<td>60</td>
<td>61</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>12</td>
<td>22</td>
<td>39</td>
<td>65</td>
<td>97</td>
<td>122</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>17</td>
<td>32</td>
<td>57</td>
<td>94</td>
<td>141</td>
<td>187</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>41</td>
<td>74</td>
<td>122</td>
<td>182</td>
<td>241</td>
<td>282</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>86</td>
<td>143</td>
<td>213</td>
<td>282</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Speed-up from parallel processing, relative to single processor for the above

<table>
<thead>
<tr>
<th>No. of Models</th>
<th>No. of Processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>8192</td>
<td>1.00</td>
<td>2.00</td>
<td>4.00</td>
<td>7.99</td>
<td>15.94</td>
<td>31.76</td>
<td>63.03</td>
<td>124.15</td>
<td>241.00</td>
<td>465.21</td>
<td>819.36</td>
<td></td>
</tr>
<tr>
<td>4096</td>
<td>1.00</td>
<td>2.00</td>
<td>4.00</td>
<td>3.99</td>
<td>7.97</td>
<td>15.88</td>
<td>31.52</td>
<td>62.09</td>
<td>120.53</td>
<td>227.65</td>
<td>409.75</td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td>1.00</td>
<td>2.00</td>
<td>4.00</td>
<td>3.99</td>
<td>7.95</td>
<td>15.77</td>
<td>31.06</td>
<td>60.29</td>
<td>113.88</td>
<td>204.96</td>
<td>341.58</td>
<td></td>
</tr>
<tr>
<td>1028</td>
<td>1.00</td>
<td>2.00</td>
<td>4.00</td>
<td>3.98</td>
<td>7.89</td>
<td>15.55</td>
<td>30.18</td>
<td>57.01</td>
<td>102.64</td>
<td>171.11</td>
<td>208.61</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>1.00</td>
<td>1.99</td>
<td>3.95</td>
<td>7.79</td>
<td>15.11</td>
<td>28.54</td>
<td>51.36</td>
<td>86.56</td>
<td>107.84</td>
<td>107.84</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>1.00</td>
<td>1.98</td>
<td>3.91</td>
<td>7.59</td>
<td>14.32</td>
<td>25.77</td>
<td>42.90</td>
<td>57.87</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>1.00</td>
<td>1.97</td>
<td>3.82</td>
<td>7.21</td>
<td>12.97</td>
<td>21.58</td>
<td>32.31</td>
<td>32.93</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>1.00</td>
<td>1.94</td>
<td>3.66</td>
<td>6.58</td>
<td>10.94</td>
<td>16.35</td>
<td>20.56</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1.00</td>
<td>1.89</td>
<td>3.39</td>
<td>5.62</td>
<td>8.39</td>
<td>11.14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1.00</td>
<td>1.79</td>
<td>2.98</td>
<td>4.44</td>
<td>5.88</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.00</td>
<td>1.66</td>
<td>2.47</td>
<td>3.27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Update Rate (Hz) predicted for a system with mobile entity models

<table>
<thead>
<tr>
<th>No. of Models</th>
<th>No. of Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>4096</td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td>0</td>
</tr>
<tr>
<td>1028</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>0</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
</tr>
<tr>
<td>128</td>
<td>2</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
</tr>
<tr>
<td>32</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td>42</td>
</tr>
<tr>
<td>8</td>
<td>88</td>
</tr>
</tbody>
</table>

### Speed-up from Parallel Processing, relative to single processor

<table>
<thead>
<tr>
<th>No. of Models</th>
<th>No. of Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>8192</td>
<td>1.00</td>
</tr>
<tr>
<td>4096</td>
<td>1.00</td>
</tr>
<tr>
<td>2048</td>
<td>1.00</td>
</tr>
<tr>
<td>1028</td>
<td>1.00</td>
</tr>
<tr>
<td>512</td>
<td>1.00</td>
</tr>
<tr>
<td>256</td>
<td>1.00</td>
</tr>
<tr>
<td>128</td>
<td>1.00</td>
</tr>
<tr>
<td>64</td>
<td>1.00</td>
</tr>
<tr>
<td>32</td>
<td>1.00</td>
</tr>
<tr>
<td>16</td>
<td>1.00</td>
</tr>
<tr>
<td>8</td>
<td>1.00</td>
</tr>
</tbody>
</table>

XXV
Appendix D    Benchmark Results on Registration and Interest Agent Prototype

<table>
<thead>
<tr>
<th>Type of Agent</th>
<th>Condition</th>
<th>No. of agents per processor</th>
<th>No. of updates per agent</th>
<th>Total time (sec) single proc.</th>
<th>dual proc.</th>
<th>Update times (sec) single proc.</th>
<th>dual proc.</th>
<th>multiple proc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodal</td>
<td>Nodes only</td>
<td>12</td>
<td>10</td>
<td>41</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
<td>20</td>
<td>60.5</td>
<td>0.16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
<td>30</td>
<td>81</td>
<td>0.17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>10</td>
<td>116</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>20</td>
<td>194</td>
<td>1.30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>30</td>
<td>275</td>
<td>1.35</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mean</td>
<td>0.17</td>
<td>1.33</td>
<td>2.48</td>
</tr>
<tr>
<td></td>
<td>Six mobile models</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nodal</td>
<td>Node+sim</td>
<td>6</td>
<td>30</td>
<td>99</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Registration</td>
<td>Node+sim+reg</td>
<td>6</td>
<td>29</td>
<td>147</td>
<td>0.28</td>
<td></td>
<td></td>
<td>t_g</td>
</tr>
<tr>
<td>Interest</td>
<td>Node+sim+reg+int</td>
<td>36</td>
<td>9</td>
<td>150</td>
<td>0.04</td>
<td></td>
<td></td>
<td>t_x</td>
</tr>
<tr>
<td></td>
<td>Twelve mobile models</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nodal</td>
<td>Node+sim</td>
<td>12</td>
<td>30</td>
<td>116</td>
<td>320</td>
<td></td>
<td></td>
<td>t_g</td>
</tr>
<tr>
<td>Registration</td>
<td>Node+sim+reg</td>
<td>12</td>
<td>26</td>
<td>200</td>
<td>0.27</td>
<td></td>
<td></td>
<td>t_x</td>
</tr>
<tr>
<td>Interest</td>
<td>Node+sim+reg+int</td>
<td>72</td>
<td>4.5</td>
<td>190</td>
<td>0.04</td>
<td></td>
<td></td>
<td>t_i</td>
</tr>
</tbody>
</table>

The interaction time is estimated by requiring that the values of t_x be consistent for six and twelve models.
Appendix E  Performance Predictions for System of Registration and Interest Agents

Update Rate (Hz) predicted from parameters found in Appendix 7-A

<table>
<thead>
<tr>
<th>No. of Models</th>
<th>No. of Processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>8192</td>
<td></td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.003</td>
</tr>
<tr>
<td>4096</td>
<td></td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.001</td>
<td>0.003</td>
<td>0.007</td>
</tr>
<tr>
<td>2048</td>
<td></td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.001</td>
<td>0.003</td>
<td>0.007</td>
<td>0.015</td>
</tr>
<tr>
<td>1024</td>
<td></td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.001</td>
<td>0.003</td>
<td>0.007</td>
<td>0.030</td>
</tr>
<tr>
<td>512</td>
<td></td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.001</td>
<td>0.003</td>
<td>0.007</td>
<td>0.015</td>
<td>0.015</td>
<td>0.015</td>
</tr>
<tr>
<td>256</td>
<td></td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.001</td>
<td>0.003</td>
<td>0.007</td>
<td>0.015</td>
<td>0.015</td>
<td>0.030</td>
</tr>
<tr>
<td>128</td>
<td></td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.001</td>
<td>0.003</td>
<td>0.007</td>
<td>0.015</td>
<td>0.015</td>
<td>0.030</td>
</tr>
<tr>
<td>64</td>
<td></td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.001</td>
<td>0.003</td>
<td>0.007</td>
<td>0.015</td>
<td>0.015</td>
<td>0.030</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td>0.001</td>
<td>0.001</td>
<td>0.003</td>
<td>0.007</td>
<td>0.015</td>
<td>0.030</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>0.006</td>
<td>0.004</td>
<td>0.011</td>
<td>0.024</td>
<td>0.049</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>0.021</td>
<td>0.017</td>
<td>0.039</td>
<td>0.079</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Speed-up from Parallel Processing, relative to single processor

<table>
<thead>
<tr>
<th>No. of Models</th>
<th>No. of Processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>8192</td>
<td></td>
<td>1.00</td>
<td>3.39</td>
<td>12.72</td>
<td>46.56</td>
<td>165.26</td>
<td>564.13</td>
<td>1836.40</td>
<td>5656.68</td>
<td>16389.38</td>
<td>44507.33</td>
<td>113096.29</td>
</tr>
<tr>
<td>4096</td>
<td></td>
<td>1.00</td>
<td>3.18</td>
<td>11.64</td>
<td>41.33</td>
<td>141.09</td>
<td>459.28</td>
<td>1414.73</td>
<td>4098.97</td>
<td>11131.25</td>
<td>28285.29</td>
<td>67121.03</td>
</tr>
<tr>
<td>2048</td>
<td></td>
<td>1.00</td>
<td>2.91</td>
<td>10.34</td>
<td>35.30</td>
<td>114.91</td>
<td>353.96</td>
<td>1025.56</td>
<td>2785.02</td>
<td>7076.93</td>
<td>16793.57</td>
<td>36986.99</td>
</tr>
<tr>
<td>1024</td>
<td></td>
<td>1.00</td>
<td>2.59</td>
<td>8.84</td>
<td>28.77</td>
<td>88.63</td>
<td>256.80</td>
<td>697.36</td>
<td>1772.04</td>
<td>4205.06</td>
<td>9261.43</td>
<td>18661.27</td>
</tr>
<tr>
<td>512</td>
<td></td>
<td>1.00</td>
<td>2.22</td>
<td>7.22</td>
<td>22.23</td>
<td>64.40</td>
<td>174.89</td>
<td>444.42</td>
<td>1054.60</td>
<td>2322.71</td>
<td>4680.13</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td></td>
<td>1.00</td>
<td>1.82</td>
<td>5.59</td>
<td>16.20</td>
<td>44.00</td>
<td>111.81</td>
<td>265.33</td>
<td>584.37</td>
<td>1177.47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td>1.00</td>
<td>1.42</td>
<td>4.10</td>
<td>11.14</td>
<td>28.31</td>
<td>67.18</td>
<td>147.95</td>
<td>298.11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td>1.00</td>
<td>1.05</td>
<td>2.86</td>
<td>7.26</td>
<td>17.22</td>
<td>37.93</td>
<td>76.43</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td>1.00</td>
<td>0.75</td>
<td>1.91</td>
<td>4.53</td>
<td>9.97</td>
<td>20.09</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>1.00</td>
<td>0.76</td>
<td>1.87</td>
<td>4.25</td>
<td>8.63</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>1.00</td>
<td>0.79</td>
<td>1.85</td>
<td>3.75</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

xxvii
## Update Rate (Hz) predicted with 200 time execution speedup

<table>
<thead>
<tr>
<th>No. of Models</th>
<th>No. of Processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>8192</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.001</td>
<td>0.003</td>
<td>0.009</td>
<td>0.028</td>
<td>0.081</td>
<td>0.220</td>
<td>0.560</td>
<td>1.329</td>
</tr>
<tr>
<td>4096</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.001</td>
<td>0.003</td>
<td>0.009</td>
<td>0.028</td>
<td>0.081</td>
<td>0.220</td>
<td>0.560</td>
<td>1.329</td>
<td>2.928</td>
</tr>
<tr>
<td>2048</td>
<td>0.000</td>
<td>0.000</td>
<td>0.001</td>
<td>0.009</td>
<td>0.028</td>
<td>0.081</td>
<td>0.220</td>
<td>0.560</td>
<td>1.329</td>
<td>2.928</td>
<td>5.904</td>
<td>9.827</td>
</tr>
<tr>
<td>1024</td>
<td>0.005</td>
<td>0.009</td>
<td>0.028</td>
<td>0.081</td>
<td>0.220</td>
<td>0.560</td>
<td>1.329</td>
<td>2.928</td>
<td>5.904</td>
<td>9.827</td>
<td>15.850</td>
<td>29.703</td>
</tr>
<tr>
<td>512</td>
<td>0.005</td>
<td>0.009</td>
<td>0.028</td>
<td>0.081</td>
<td>0.220</td>
<td>0.560</td>
<td>1.329</td>
<td>2.928</td>
<td>5.904</td>
<td>9.827</td>
<td>15.850</td>
<td>29.703</td>
</tr>
<tr>
<td>256</td>
<td>0.020</td>
<td>0.028</td>
<td>0.081</td>
<td>0.220</td>
<td>0.560</td>
<td>1.329</td>
<td>2.928</td>
<td>5.904</td>
<td>9.827</td>
<td>15.850</td>
<td>29.703</td>
<td>59.407</td>
</tr>
<tr>
<td>128</td>
<td>0.294</td>
<td>0.220</td>
<td>0.560</td>
<td>1.329</td>
<td>2.928</td>
<td>5.904</td>
<td>9.827</td>
<td>15.850</td>
<td>29.703</td>
<td>59.407</td>
<td>118.815</td>
<td>237.630</td>
</tr>
<tr>
<td>32</td>
<td>1.141</td>
<td>0.859</td>
<td>2.133</td>
<td>4.839</td>
<td>9.827</td>
<td>15.850</td>
<td>29.703</td>
<td>59.407</td>
<td>118.815</td>
<td>237.630</td>
<td>475.261</td>
<td>950.521</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Update Rate (Hz) predicted with 200 time execution speedup and 10 times as many interested entities

<table>
<thead>
<tr>
<th>No. of Models</th>
<th>No. of Processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>8192</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.001</td>
<td>0.003</td>
<td>0.009</td>
<td>0.028</td>
<td>0.081</td>
<td>0.220</td>
<td>0.560</td>
<td>1.329</td>
</tr>
<tr>
<td>4096</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.001</td>
<td>0.003</td>
<td>0.009</td>
<td>0.028</td>
<td>0.081</td>
<td>0.220</td>
<td>0.560</td>
<td>1.329</td>
<td>2.928</td>
</tr>
<tr>
<td>2048</td>
<td>0.000</td>
<td>0.000</td>
<td>0.001</td>
<td>0.009</td>
<td>0.028</td>
<td>0.081</td>
<td>0.220</td>
<td>0.560</td>
<td>1.329</td>
<td>2.928</td>
<td>5.904</td>
<td>9.827</td>
</tr>
<tr>
<td>1024</td>
<td>0.005</td>
<td>0.009</td>
<td>0.028</td>
<td>0.081</td>
<td>0.220</td>
<td>0.560</td>
<td>1.329</td>
<td>2.928</td>
<td>5.904</td>
<td>9.827</td>
<td>15.850</td>
<td>29.703</td>
</tr>
<tr>
<td>512</td>
<td>0.005</td>
<td>0.009</td>
<td>0.028</td>
<td>0.081</td>
<td>0.220</td>
<td>0.560</td>
<td>1.329</td>
<td>2.928</td>
<td>5.904</td>
<td>9.827</td>
<td>15.850</td>
<td>29.703</td>
</tr>
<tr>
<td>256</td>
<td>0.020</td>
<td>0.028</td>
<td>0.081</td>
<td>0.220</td>
<td>0.560</td>
<td>1.329</td>
<td>2.928</td>
<td>5.904</td>
<td>9.827</td>
<td>15.850</td>
<td>29.703</td>
<td>59.407</td>
</tr>
<tr>
<td>128</td>
<td>0.294</td>
<td>0.220</td>
<td>0.560</td>
<td>1.329</td>
<td>2.928</td>
<td>5.904</td>
<td>9.827</td>
<td>15.850</td>
<td>29.703</td>
<td>59.407</td>
<td>118.815</td>
<td>237.630</td>
</tr>
<tr>
<td>32</td>
<td>1.141</td>
<td>0.859</td>
<td>2.133</td>
<td>4.839</td>
<td>9.827</td>
<td>15.850</td>
<td>29.703</td>
<td>59.407</td>
<td>118.815</td>
<td>237.630</td>
<td>475.261</td>
<td>950.521</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

xxviii
Appendix F Pseudo Code for Discovery and Object Mobility

Contents of Nodal Information

Name of Node (unique), e.g. n1, n2, n3 etc.

Bi-directional links to neighbouring nodes, with unique names,

e.g. n1-n2, n1-n3, n2-n3 etc.

Home node: link "+h" leads to a node with content:

Local time, local position, no. of registration agents

Neighbours node: link "+n" leads to a node with content:

Name of link, name of first neighbour, local time, position, no. of agents, ....

Simulator node (optional): link "+s" leads to a node with content:

Time, free processing power, name of first model, name of 2nd model,.....

Registration node: links "+r" lead to one of several registration nodes, with contents:

Time of arrival, ident. of entity, Position, Signature information

From these, temporary links "+i" lead to interest declaration nodes, with content:

Time of registration, ident. of entity, site address, information spec.

Simulation Interface Processing

Initially:

Establish contact with the nodal process and set up a "+s" node attached to it with contents:

Time, free processing power, list of model types available on this simulator.

Loop:

For each local entity:

Chose a Universal Identifier for it - not tied to a particular simulator.
Pseudo-Code for Agents

Check that its Registration Agent is still active - if not cause a new one to be activated with universal id, launch time, signature information, local address.

Send position & attitude of entity to Registration Agent.

Send any environmental effect produced to Registration Agent.

Receive list of interested entities, addresses and information specs (type and update rate of information) from Registration Agent.

Set up communications paths to interested entities and send requested information to them.

Receive instruction to move entity to new site - launch migration agent with state data.

For each entity migrated away:

Continue to generate data locally until model is established.

Monitor correct generation of state data from new site.

If correctly operating then

Close down local entity after a time-out period

else

Send a migration agent to abort entity on new site

Resume normal processing of entity locally.

end if

For each remote entity:

Receive information sent by entity and distribute among local entities.

Receive a migration agent, and create a new local model, initialised with state data received or abort operation of an unsuccessfully migrated model.

Find out how much processing power is still unused on this node, and advertise it in the "+s" node, with an updated time stamp.
Pseudo-Code for Agents

Nodal Processing

Initially:
Set up an "h" node.

Visit neighbours - find node names and links. Set up an "n" node.

Loop:

Publish current position and local time on "h" node.

Go to neighbouring nodes; extract positions, times, names and link-names. If any nodes or links are missing, initiate node structure recovery. Insert positions & times into "n" node.

Go to all "r" nodes; count them and extract arrival times & goal positions.

Calculate new local time:

If a simulator is present, then

take its time as your own,

else advance own time by an appropriate amount

end if

Find mean of neighbours' times.

if Own_time > Mean_time + time_tolerance then

Own_time = Mean_time + time_tolerance

else if Own_time < Mean_time - time_tolerance then

Own_time = Mean_time - time_tolerance

end if

If not an edge node, calculate new position for node:

If more than reg_threshold agents have each been present for more than reg_time,
then

Move towards mean of goals by move_fraction

xxx
Limit move to lie within bounding box of neighbours

else

Find mean position of neighbours - move towards it by move_fraction

end if

Registration Agent

Loop:

Receive updates from originating simulation giving position and attitude and changes to signature data of entity. Also any environmental effects produced by entity.

Optimise position with respect to the centres of interest of the processing nodes.

On moving to a new node, create an "+r" node with arrival time, goal position.

If an environmental effect received, communicate it to the appropriate environmental agent.

Look through list of already known interested entities, removing any which have timed out.

Look for "i" nodes from other interest agents - extract entity id and add to list of interested entities. Remove "i" node.

Every interest_time seconds, launch an interest agent to search for detectable entities.

Look through list of transmitting entities - remove any which have timed out. Receive information from own interest agent about entities which will transmit to this one.

Every site_search_time, launch a site search agent to look for candidate sites for model migration.

Estimate communications cost for current model site using lists of interested and transmitting entities.

Estimate communications costs and position errors for new sites sent by search agents.

When all site information returned, choose best site for model execution. If it is a new site, send location to originator.
Registration Agent Navigation Algorithm

Find error vector between goal position and all neighbouring nodes (including current node).

If any neighbour has a smaller error than the current one, and the error vector is in the same direction (within 90°), then

move to it.

else if a neighbour has a smaller error, but the error vector is not in the same direction (angle > 90°), then

move only if the error is smaller by a factor "hop_threshold".

else

stay put, but periodically perform the same algorithm using the errors resulting from two hops, to avoid local minima.

end if

The Registration Agent also checks that the number of agents currently on a node is less than some maximum value "Max_Registrars" before it makes a move.

If this results in it being stalled with a large error magnitude for longer than a certain period of time, it will attempt a "sideways hop", moving to an arbitrary node roughly perpendicularly to its error vector.

Interest Agent

Create a spanning tree, spreading out from registration agent site. Depending on algorithm given to agent, prune tree by direction, or by distance, or by environmental values (e.g. transmittance) or by entity detections.

At each site, inspect each registration agent. Use algorithm given to decide on detectability, depending on signature of entity, position, attitude, and values of environmental variables (possibly integrated from here to sensor).

If detected, add own identity and address and time to a "+i" node attached to the "+r" node of the registration agent. Also send information back to originating registration agent about it.
Psuedo-Code Relevant to Site Search and Mobility Agents

Registration Agent Interaction with Site Search and Mobility Agents

Accumulate list of interested entities and list of entities in view (with positions and positions of their simulators, and estimate of data traffic to each (allowing for multi-casting arrangements)).

Compute:

a) the cost of communications from current site, and

b) the geographical match.

If

geographical match < threshold1,

or

communication cost < threshold2,

then stop.

Find out what type of simulation resources are needed by own model.

Provided that a move of self to another node is not imminent:

Dispatch a site search agent to look for suitable simulation resources.

Receive replies from the site search agents. For each reply, compute

a) the cost of communications from new site.

b) the geographical match for new site.

If

comms cost < current comms cost - threshold1

and

geo. match < current geo. match - threshold2

then
Pseudo-Code for Agents

enter new site on potential sites list

end if

if potential sites list not empty then

for all sites on potential sites list

find minimum of (comms cost * w1 + geo. cost * w2)

end for

Now have selected a new site.

end if

if new site was selected then

Send message to new site, reserving the resources required.

Send message to simulator, telling it where to move entity.

Add new site to interest lists of all entities on in-view list for own entity.

end if

Site Search Agent

Create a spanning tree, spreading out from registration agent site. Depending on algorithm given to agent, prune tree by direction, or by distance, or by number of suitable sites seen.

Maximum distance should not exceed distance from Registration Agent to current site.

Find execution sites with suitable models by inspecting their "+s" node contents. Also ensure that the node has sufficient processing resources to accept a new model.

Send location of new site back to Registration Agent, and continue.

Mobility Agent

Hop from old site to new site, or from Set-up to initial site:

either carrying initialisation information for creation of a new instance,
Pseudo-Code for Agents

or carrying state information to continue the existence of an old instance on a new site, dead-reckoning state information up to time of restarting

or carrying the instruction to stop execution of an instance at a site.

Old Simulator Site

Send states of entity to registration agent.

Start mobility agent when requested.

Monitor move:

   Inspect states from new and old executions -

   if they agree reasonably well for \( n \) samples, then:

       kill own model, and send an interest list to the new model.

   else

       send a kill message to the new model, retry move at least once more.

   end if

Monitor operation of registration agent

   - start a new one if necessary.

New Simulator Site

Post a list of modelling resources present and available for use.

Accept a message reserving resources, and reply with an acknowledgement.

Allow mobility agent to run new model.

Send states of entity to registration agent.

Accept an interest list for the new model.

Kill new model when requested.
Appendix G  WAVE code for Registration and Interest Agents Prototype

```

Filename: node
File Description : nodal processing for relevance filtering
Author: M. J. Corbin
File Creation Date: 27.11.95
Compiler/Language: WAVE

(DN /.. $# DN), . FT='node'.FTC.T=FT.

CL{ node # ` 
\ parameters of process
Ptinc = 5. \ time increment value
Ftime = 20. \ time after which registration agent is treated as static
Fmfr = 4. \ move fraction
Frthr = 2. \ number of registration agents needed to trigger movement
\ Fedge will be set to 1 if this is an edge node
\ definition of operations carried out on position information
Fpsize = 2. \ number of elements in a position record
Fpos = 'Fx = 0. Fy = 0'.
Fgpos = 'Fp = Fx. Fp % Fp & Fy'.
\ zero position counters
Fpszero = 'Npsum = 0. Nx = 0. Ny = 0'.
\ add up positions
\ find the average goal position of registration agents camped on this node
Fpave = OS( (Npsum > 0. Nx / Npsum. Ny / Npsum), ) '.
Fmove = 'Nx - Fx. Nx / Fmfr. Fx + Nx. Ny - Fy. Ny / Fmfr. Fy + My '.
Flimit = ' '.
Fplan = 'FG = 'object_plan'.
```

xxxvii
Prototype Code

\[ F_{xl} = F_x \cdot F_{xl} \cdot F_{scl} \cdot F_{xl} + F_{xc} \cdot F_G + F_{xl}. \]
\[ F_{yl} = F_y \cdot F_{yl} \cdot F_{scl} \cdot F_{yl} + F_{yc} \cdot F_G + F_{yl} \cdot F_G \cdot C \cdot F_G \% - \'_'. \]
\[ G = F_G \cdot !3. \]

\[ \text{\textbackslash find mean position of neighbouring nodes} \]
\[ F_{mneigh} = \text{Fpszero}. \rightarrow \text{n \#}. \quad FT = \text{C.} \quad FT \mid \text{'}, \text{'.} \quad \text{Faddr.} \quad Fi = 4. \]
\[ \text{RP} \{
\quad F_p = FT \cdot F_{pl} = Fi \cdot Fpsum \cdot F_i + 4. \quad F_i + Fpsum. \n\}, \quad \text{Fpave} \]

\[ \text{\textbackslash set initial variables} \]
\[ \text{Faddr = A. OS( \{ Fedge = Fzpos \}, ).} \quad \text{Ftime = 0.} \quad \text{Front = 0.} \]

\[ \text{\textbackslash create a home node initially} \]
\[ \text{FT = `0'.} \quad \text{Fzpos.} \quad \text{FT \& Fp.} \quad \text{FT \& `0'.} \]
\[ \text{CL}\{
\quad \text{h \# home.} \quad \text{FT \& ',}. \quad \text{C = FT.} \quad !3
\}

\[ \text{\textbackslash find neighbouring nodes and links} \]
\[ \text{NLink = . Nears = NT = .} \]
\[ \text{SQ} \{
\quad \text{\# A. \#.} \quad \text{Fl = L.} \quad \text{Fl /= h.} \quad \text{Fl /= r.} \quad \text{Fl /= e.} \quad \text{Fc = C.} \]
\quad \text{\# Faddr.} \quad \text{Nlink \& Fl.} \quad \text{Nears \& Fc.} \quad \text{NT \& Fl.} \quad \text{NT \& Fc.} \quad \text{NT \& FT.} \quad !3
\}, \quad \text{\textbackslash create a neighbours node describing them} \]
\[ \quad \text{CL}\{
\quad \quad \text{FT = NT.} \quad \text{FT \& ',.} \quad \text{h \# neighbour.} \quad \text{C = FT.} \quad !3
\}

\[ \text{\textbackslash find simulator, and publish a list of available models} \]

\[ \text{\textbackslash diagnostic print out} \]
\[ \text{SQ(\{ NT`\textit{initially:}_'. \#A.\#. FT=S.FT&L.FT&C.\#Faddr.NT&FT.\!3\}, (NT`\_'. T=NT)\}). \]

\[ \text{\textbackslash main iteration loop starts here} \]
\[ \text{RP}\{
\quad \text{\textbackslash advance local time} \]
\quad \text{\# A. Ftime + Ftinc.} \]
\quad \text{\textbackslash publish local time and position of centre of interest} \]
\[ \text{SQ} \{
\quad \text{FT = Ftime.} \quad \text{FT \& Fplan.} \quad \text{Fzpos.} \quad \text{FT \& Fp.} \quad \text{FT \& Front.} \quad \text{FT \& ',.} \quad \text{h \#.} \quad \text{C = FT.} \quad !3
\}, \quad \text{\textbackslash xxxviii} \]
update information about neighbouring nodes

```
NT = .
SQ(
    ( Nlink #. FT = L. FT & C. +h #. FT & C. @@ Faddr. NT & FT. !3
    ),
    ( FT = NT. FT % ~, `. +n #. C = FT. @@ Faddr
    )
).
```

check continued existence of neighbouring nodes and links

adjust local time estimate to line up with neighbours

```
Ftmean = 0. Ftcnt = 0. Fi = 3. NT | ~, '

\ find max and min times among neighbours
RP(
    PT = NT. FT : Fi. FT / - . Ftmean + PT. Ftcnt + 1. Fi + 4. Fi + Fpsize
).

\ adjust local time to lay between Ftmax+fptinc and Ftmin-fptinc
OS(
    ( Ftcnt > 0. Ftmean / Ftcnt.
    OS(
        ( Ftime > Ftmax. Ftime = Ftmax
        ),
        ( Ftime < Ftmin. Ftime = Ftmin
    )
).
).
```

find all registration agents camped on this node

```
NT = 0. Fpszero. Ftr = Ftime. Ftr - Ftime.
SQ(
    \ count them
    ( +r #. FT = C. FT | ~, `. @@ Faddr. NT + 1.

\ find ones which have been here more than Ftrtime
    FT = FT. Ft : 1. Ft < Ftr.

\ find mean goal position of those
    Fp = FT. Fpl = 3. Fpsum. !3
    ( Front = NT.
).
```

find new position for centre of interest of this node

```
OS(
    \ if this is an edge node, do not move
    ( Edge /-
    ),
    \ move towards mean of registration agents' positions,
    \ limited by bounds set by neighbours' positions.
    ),
```

xxxix
or move towards mean of neighbours' positions
   ( Fmneigh, Fmove
   )
).

check on continued updating of any environmental information
   SQ(
      { +e #, FT = C, FT | ` `, FT : l, FT < Ftr, l3
      },
   ).

\ diagnostic print out
\SQ((NT=`loop: ` .@#A. #. FT=S, FT&L, FT&C. @#Faddr, NT&FT. l3), (NT%= ` .T=NT)).
)
)
)

Prototype Code

```plaintext
Filename: reg
File Description : registration agent for relevance filtering
Author: M. J. Corbin
File Creation Date: 4.12.95
Compiler/Language: WAVE

(DN /-. @# DN), FT='reg'.FT&=FT.
CL(' reg #`

number of elements in a position record
Fpsize = 2.

spawn a new interest agent
Fsint = ` @# code, int #. Fint = C. @# Phreg. Fint '

time interval between interest agent launches
Fint = 10.

time interval between site search agent launches
Fsrch = 75.

maximum number of agents allowed on any one node
Fmaxag = 2.

extract agent goal position from a string and put it into Fx, Fy

append agent goal position to Fp
Fgpox = ` Fp & Fx. Fp & Fy '

extract nodal centre of interest position and leave in Fhx, Fhy

find error vector Fhex, Fhey between node position and goal, and magnitude^2 Fher
     Fp = Fhey. Fp * Fp. Fher + Fp

find the error vector Fhex, Fhey and magnitude^2 Fher between a neighbour and goal
     Fner = Fnex. Fner * Fner. FT = Fney. FT * FT. Fner + FT

Remove a timed-out entry in the list of interested entities

xli
```
Prototype Code

\draw position of agent on schematic Network display

\begin{verbatim}
Fnet = 'FG = `plane_net', FG & Fid. FG & Fpn. FG & C. FG & `3'. FG & `Red'. FG % `.
SQ( ( @# Fanet. G = FG. !3 ), )
end.
\end{verbatim}

\draw an arrow showing detection of an entity on network graph

\begin{verbatim}
Finf = 'FT = `inform_net'. FT & Fho. FT & Fco. FT & 1. FT & 2. FT & `Red'. FT % `.
SQ( ( @# Fanet. G = FT. !3 ), )
end.
\end{verbatim}

set variables initially

Fr = Frig = A. Fint = 0. Ftsrch = 0. Freg = 0. Phreg = A. Fpn = C. Fnet.

\main iteration loop
\begin{verbatim}
RP(
\receive update of goal position from originating simulation and change "+r" node
\begin{verbatim}
OS(
)
).
\repeat the migration algorithm until no move is possible
RP(
\get position and time of local processing node and find error from goal
Fhome. Fhev.
\decide whether to re-optimise position with respect to processing nodes
+1 %. Ft = C. @# P. Ft | ``,`. Fpi = 0. Ferrmin = Fher. Fbest = .
OS(
( Fnum < Fmaxag. Fner < Ferrmin. Ferrmin = Fner. Fbest = Fl
)
).
\make a move if a better node has been found
Fbest /= .
FT=Moving'.FT&Fid.FT&from : .FT&C.FT&via : .FT&Fbest.FT&Ferrmin.FT % `.T=FT.
SQ(
( Freg /= 0. @# Freg. C = .!3 \ first remove any "+r" node
)
).
Fbest #.
Freg = 0. Phreg = A. Fnet. Fpn = C
).
\get information from home node again (lost when repeat loop above failed)
Fhome.
\if a move has just been made, create a "+r" node
OS(
( Freg == 0. Fp = Ftime. Fp & Fid. Fppos.

xliii

\end{verbatim}

xliii
Prototype Code

CL(+x # reger. Freg = A. Fp % ',', C = Fp)
#
## Freg

\ every interest_time: ...

OS{
    Ftime > Fint. Fint + Fint.
    \ inspect list of transmitting entities, and remove any timed out
    Fi = 0.
    RP{
        Fi + 1. Ft = Ftime. Ft : Fi. Ft /-. Ft + 0. OS{ Ft < Ftime. Ftrim },
    }
    \ look for information sent back about transmitting entities
    RP{
        Ntrans /-. Ftins
    }
    \ inspect list of interested entities, and remove any timed out
    Fi = 0.
    RP{
        Fi + 1. Ft = Nitime. Ft : Fi. Ft /-. Ft + 0. OS{ Ft < Ftime. Firem },
    }
    \ look for "*i" nodes indicating interest from other entities
    SQ{
        SQ(+i #. Fp = C. SQ( @# F, ( C = !3 ) ). Ftins. !3
    }
    \ send back list of interested entities to originating simulation
    Print,
    \ launch a fresh interest agent to look for entities
    { Fsmart
    },
    \ otherwise, fetch a position update
    SQ( Fpos, )
    }
    \ every search_time, decide whether to move entity to a new site
    OS{
        Ftime > Fsrch. Fsrch + Fsrch.
        \ and if not, launch a fresh site search agent to look for new sites
        Fsrch,
    },
    @# Fhreg
}

Filename: interest

File Description: interest agent for relevance filtering

Author: M. J. Corbin

File Creation Date: 4.12.95

Compiler/Language: WAVE

---

\( (DN/-@#DN)/.,\ FT='int'.FT&C.T=FT.\)

CL( @# code. int # `)

---

number of elements in a position record

Fpsize = 2.

detection routine - currently detects everything bar oneself

Fdet = ` Fn = C. Fn | `.' Fn : 2. Fn /= Fid. Fn & A. Fn & Fco. Fn & Frate.
       Fp = Fid. Fp & Forig. Fp & Frate. Fp & `.'
       CL( +i # int. C = Fp ). @# Freg. Ntrans & Fn '.

---

prune the search tree - currently does just a few hops

Flimit = 2.

Fprune = ` Fc < Flimit. Fc + 1 '.

---

recursive search for registration agents

Fspread = `.

move to registration agents' nodes and see if entities are detectable
  ( Fco = C. +r #. Fdet. !3
 ),

prune search tree of processing nodes
Fprune.

get list of links to neighbouring nodes
+n #. Fp = C. @# F. Fp | `.' Fi = 1. Fl =.
RP(
    Ft = Fp. Ft : Fi. Ft /= Fl & Ft. Fi + 4. Fi + Fpsize
 ),

hop to neighbouring nodes and mark them as a tree structure
SQ( Fl #).

mark each node with new interest agent time to colonise it
  Ftst
 ).

continue this process recursively
Fspread

---
Prototype Code

\ write a routine to test and set a distinct nodal variable: N<Fid> /= Ftime. N<Fid> = Ftime

FWrit = `Fst = 'N'. Fst & Fid. Fst & `/=Ftime.' . Fst & '"'. Fst & Fid. Fst & "=Ftime'. Fst& `'.

\ start a new interest agent

Frate = 10. Fc = 0. FWrit. Fst. Fspread
Filename: sim

File Description: simulate entity and launch registration agent

Author: M. J. Corbin

File Creation Date: 27.11.95

Compiler/Language: WAVE

(DN/-.@#DN), FT = 'sim'. FT & C. T = FT.

CL( sim # ~

draw the model on the network diagram


FG = 'object_net'. FG & Fnx. FG & Fny. FG & Fid. FG & 'Blue'. FG % ' ~'.

SQ( ( @# Fanet. G = FG. !3 ), )

draw model's position on plan view screen

Fdmp = 'FG = 'object_plan'.

Fx1 = Fx. Fx1 * Fscl. Fx1 + Fxc. FG & Fx1.

Fy1 = Fy. Fy1 * Fscl. Fy1 + Fyc. FG & Fy1.

FG % ,

SQ( ( @# Fanet. G = FG. !3 ), )

Fcc + 1. OS( ( Fcc > 20. Fcc = 0. Fcol = 'Blue' ), )

collect graphics information from 'gra' node

Fsim = Fcol = 'Blue'.

gra #. FG = C. FG | ^', '.

Fnet = FG. Fnet : 1.

Fplan = FG. Fplan : 2.

Fxc = FG. Fxc : 3. Fxc + 0.

Fyc = FG. Fyc : 4. Fyc + 0.

Fscl = FG. Fscl : 5. Fscl + 0.

Fg = @# P.

(DN/-.@#DN), C = Fnet. Fanet = A. \ get address of network display

(DN/-.@#DN), C = Fplan. Faplan = A. \ get address of plan view display

\ collect code for registration agent and launch it

(DN/-.@#DN), @# Forig. @# P. reg #. Frig = C. @# Forig.

Nupdat = Fx. Nupdat & Fy.

Frg,

\ draw the simulation model on the network diagram

Fdrm,

\ change position of entity with time (circle)
Prototype Code

RP(
  Fm + 1.
OS(
  { Fm > 5. Fm = 0.

    Fdx = Fx. Fdx - Fxctr. Fdx / Fdt. Fx + Fdx.
    Fdy = Fy. Fdy - Fxctr. Fdy / Fdt. Fy - Fdy.

    Nupdat = Fx. Nupdat & Fy. Fdrp
  },
  ).
)
Fi + 1.
OS(
  Fi > 10. Fi = 0.

    Niname /= 'Green'. FT = 'Sim_Receivers_for'. FT & Fid. FT & Niname.
    Niname = . FT & Nirate. FT % '_', T = FT
  ),
)
);
Appendix H    Documentation for Java Operator Interaction Agent

This documentation was generated directly from the Java source code using the “javadoc” utility from Sun Microsystems.
### Class InstanceAgent

```java
public class InstanceAgent
    extends Frame
```

**InstanceAgent.java**

**Document:** DRA/AS/SID/505/SD97001/1.0  
**Project:** Generic Agents  
**Author:** M J Corbin  
**Location:** Systems Integration, DERA, Farnborough  
**Description:** Presents a GUI Frame for monitoring and controlling one instance of a class of remote models.  
**Language:** Java  
**Issue History:**  
1.0 1/2/97 MJC First issue.

---

**Constructor Index**

- Develop InstanceAgent(int, String, InstanceButton, PrintStream)  
  constructor

**Method Index**

- action(Event, Object)  
  entry point for handling push button controls  
- addItem(AbstractItem, ItemPanel)  
  enters the fields for a model parameter into a panel and hash tables  
- destroy()  
  remove the InstanceAgent and its windows  
- extractDataForItem(StringTokenizer, double)  
  read in the keyword and value for an item  
- handleEvent(Event)  
  handler for the WINDOW_DESTROY event  
- name()  
  returns the name of this agent’s model
number()
    returns the identifier number for this agent's model

readInstanceSpec(DataInputStream)
    reads the configuration information for this model from the stream

readItemSpec(String, String, ItemPanel)
    interpret the description for one model parameter

receiveData(String)
    read in a string of data from the model

sendDataToGraph(double)
    send values of all selected parameters to the graph applet

setStatus(String, Color, Color)
    alter the current status of this instance, with foreground and background colours

wake()
    forces the agent to redraw itself

Constructors

public InstanceAgent(int num, String name, InstanceButton button, PrintStream request)

    constructor

Parameters:
    num - identifying number of the model instance
    name - name of the model instance
    button - the selector button for this model in the ClassAgent window
    request - the stream through which the model can be addressed/controlled

Methods

number

public int number()

    returns the identifier number for this agent's model

name

public String name()

    returns the name of this agent's model

wake

public synchronized void wake()

    forces the agent to redraw itself
setStatus

public void setStatus(String status,
Color fore,
Color back)

alter the current status of this instance, with foreground and background colours

destroy

public synchronized void destroy()

remove the InstanceAgent and its windows

handleEvent

public boolean handleEvent(Event event)

handler for the WINDOW_DESTROY event
Overrides:
handleEvent in class Component

action

public boolean action(Event event,
Object obj)

entry point for handling push button controls
Overrides:
action in class Component

readInstanceSpec

public synchronized void readInstanceSpec(DataInputStream in)

reads the configuration information for this model from the stream

readItemSpec

protected void readItemSpec(String token,
String s,
ItemPanel panel)

interpret the description for one model parameter
Parameters:
token - string identifying the type of the parameter
s - string containing the rest of the line
panel - the panel which will contain the parameter's fields

addItem

protected void addItem(AbstractItem item,
ItemPanel panel)

enters the fields for a model parameter into a panel and hash tables
receiveData

public synchronized void receiveData(String s)
    read in a string of data from the model

extractDataForItem

protected void extractDataForItem(StringTokenizer t, double time)
    read in the keyword and value for an item

sendDataToGraph

protected void sendDataToGraph(double time)
    send values of all selected parameters to the graph applet
Class ItemPanel

class ItemPanel
extends Panel

Document: DRA/AS/SID/505/SD97002/1.0
Project: Generic Agents
Author: M J Corbin
Location: Systems Integration, DERA, Farnborough
Description: Object class defining a panel which can contain
a number of states and parameters of models.
Language: Java
Program File: InstanceAgent.java
Issue History:
1.0 1/2/97 MJC First issue.

Variable Index

* button
  panel’s selector button
* items
  vector of AbstractItems contained within panel
* name
  name of panel
* number
  identifying number
* repeating
  true when the panel is being updated regularly

Constructor Index

* ItemPanel(int, String)
  constructor

Method Index
- `getPanel(int, Vector)`
  finds an ItemPanel with a particular number within a vector

**Variables**

- `number`
  public int number
  identifying number

- `name`
  public String name
  name of panel

- `repeating`
  public boolean repeating
  true when the panel is being updated regularly

- `button`
  public Button button
  panel’s selector button

- `items`
  public Vector items
  vector of AbstractItems contained within panel

**Constructors**

- `ItemPanel`
  public ItemPanel(int n, String s)

  constructor

  **Parameters:**
  - n - panel number
  - s - panel name

**Methods**
getPanel

public static ItemPanel getPanel(int n,
        Vector v)

        finds an ItemPanel with a particular number within a vector

Parameters:
        n - number of panel to be found
        v - vector containing all panels of interest

Returns:
        ItemPanel with corresponding number
Class AbstractItem

```
java.lang.Object
  +--- java.awt.Component
    +--- java.awt.Container
      +--- java.awt.Panel
        +--- AbstractItem
```

public class AbstractItem
extends Panel
implements Enumeration

Document: DRA/AS/SID/505/SD97003/1.0
Project: Generic Agents
Author: M J Corbin
Location: Systems Integration, DERA, Farnborough
Description: Abstract Class providing procedure definitions for items which can be states or parameters of a model in various formats.
Language: Java
Program File: AbstractItem.java
Issue History:
1.0 1/2/97 MJC First issue.

Variable Index

- **STALE_TIME**
time after which value sent by model becomes stale and is not graphed
- **beingGraphed**
true if this item is connected to a graph
- **canAlter**
true if the operator can alter the value of this item
- **myButton**
button used to enable and disable graphing for this item
- **myKey**
keyword for items which have only one
- **myLabel**
name used to identify this item on panel and on graph
- **myValue**
current value of item sent by model
- **timeTag**
time for which the current value from the model is valid
- **valueDisplay**
label used to display value of item sent by model
AbstractItem

Method Index

- action(Event, Object)
  handles button presses to enable and disable graphing for this item
- findLabel(String)
  recognise a label enclosed in quotes and returns rest of string.
- getData()
- graphOutput(PrintStream, double)
  write data out to the graph applet in the form "name=value"
- hasMoreElements()
- isAlterable()
- isAltered()
- keys()
  this enumeration interface is provided to allow for classes of item with more than one key
- nextElement()
- putData(String, StringTokenizer, double)
  inserts a new data value from the model into the item display

Variables

- STALE_TIME
  protected static double STALE_TIME
  time after which value sent by model becomes stale and is not graphed

- myButton
  protected Button myButton
  button used to enable and disable graphing for this item

- myLabel
  protected String myLabel
  name used to identify this item on panel and on graph

- valueDisplay
  protected Label valueDisplay
  label used to display value of item sent by model
myValue

protected String myValue

current value of item sent by model

myKey

protected String myKey

keyword for items which have only one

timeTag

protected double timeTag

time for which the current value from the model is valid

canAlter

protected boolean canAlter

true if the operator can alter the value of this item

beingGraphed

protected boolean beingGraphed

true if this item is connected to a graph

**Constructors**

AbstractItem

public AbstractItem()

**Methods**

findLabel

public String findLabel(String line)

recognise a label enclosed in quotes and returns rest of string. If no quoted label is found,
takes label from first word, returns whole string.

dkeys

public Enumeration keys()

this enumeration interface is provided to allow for classes of item with more than one key

hasMoreElements
public boolean hasMoreElements()

    Returns:
    true if keyword enumeration not exhausted

**nextElement**

public Object nextElement()

    Returns:
    next key in keyword enumeration

**putData**

public void putData(String key, StringTokenizer t, double time)

    inserts a new data value from the model into the item display
Parameters:
    key - contains the keyword for this item
    t - contains the rest of the line
    time - timetag for this data

**getData**

public String getData()

    Returns:
    a string containing any data altered by the operator

**isAlterable**

public boolean isAlterable()

    Returns:
    true if this item can be altered by the operator

**isAltered**

public boolean isAltered()

    Returns:
    true if the data for this item has been altered by the operator

**action**

public boolean action(Event event, Object obj)

    handles button presses to enable and disable graphing for this item
Overrides:
    action in class Component
GraphOutput

public synchronized void graphOutput(PrintStream graph, double atTime)

write data out to the graph applet in the form ",name=value"

Parameters:
  graph - PrintStream piped to the graph applet
  atTime - time for which graph point is being drawn
public class ClassAgent
extends Frame

Document: DRA/AS/SID/505/SD97004/1.0
Project: Generic Agents
Author: M J Corbin
Location: Systems Integration, DERA, Farnborough
Description: Presents a GUI Frame for monitoring all instances of a class of remote models.
Language: Java
Program File: ClassAgent.java
Issue History:
1.0 1/2/97 MJC First issue.

**Constructor Index**

* ClassAgent(String, PrintStream, ClassApplet)

**Method Index**

* action(Event, Object)
  handles button presses for refresh, repeat, stop
* destroy()
  destroys this window and all its InstanceButtons and inform applet
* distributeInput(String)
  interpret a string containing a reply from the remote model class
* handleEvent(Event)
  handles the WINDOW_DESTROY event
* setUpColorHash()
  set up hash table for converting strings in the config file into Colors
* setUpColors(String, Color, Color)
  insert foreground and background colours into status hash tables
* startRepeats()
request model class to start sending repeats of instance list data

stopRepeats()
request model class to stop sending repeats of instance list data

**Constructors**

- **ClassAgent**

  ```java
  public ClassAgent(String name, PrintStream simStream, ClassApplet app)
  ```

  **Parameters:**
  - name - name of this class and panel
  - simStream - stream accepting commands to remote model class
  - ClassApplet - applet which started up this class agent

**Methods**

- **distributeInput**

  ```java
  public synchronized boolean distributeInput(String inString)
  ```

  interpret a string containing a reply from the remote model class

- **handleEvent**

  ```java
  public boolean handleEvent(Event event)
  ```

  handles the WINDOW_DESTROY event
  **Overriderse:**
  handleEvent in class Component

- **action**

  ```java
  public boolean action(Event event, Object obj)
  ```

  handles button presses for refresh, repeat, stop
  **Overriderse:**
  action in class Component

- **startRepeats**

  ```java
  public synchronized void startRepeats()
  ```

  request model class to start sending repeats of instance list data

- **stopRepeats**

  ```java
  public synchronized void stopRepeats()
  ```
request model class to stop sending repeats of instance list data

- **destroy**

  ```java
  public synchronized void destroy()
  ```

  destroys this window and all its InstanceButtons and inform applet

- **setUpColors**

  ```java
  public void setUpColors(String status,
  Color foreground,
  Color background)
  ```

  insert foreground and background colours into status hash tables

- **setUpColorHash**

  ```java
  public static void setUpColorHash()
  ```

  set up hash table for converting strings in the config file into Colors
Class InstanceButton

```java
java.lang.Object
   \--- java.awt.Component
      \--- java.awt.Button
         \--- InstanceButton
```

class InstanceButton
extends Button

Document: DRA/AS/SID/505/SD97005/1.0
Project: Generic Agents
Author: M J Corbin
Location: Systems Integration, DERA, Farnborough
Description: Presents a button for launching an InstanceAgent
Language: Java
Program File: ClassAgent.java
Issue History:
1.0 1/2/97 MJC First issue.

Constructor Index

* InstanceButton(int, String, String, ClassAgent)

Method Index

* action(Event, Object)
  handle presses of this button
* clearMark()
  clear the destruction mark
* destroy()
  actually destroys the button and any instance agent
* destroyed()
  message from class agent instructing a button and its agent to vanish if still marked for destruction
* getAgent()
* getGraph()
* getName()
* getNum()
* getStatus()
* makeAgent()
  launch an instance agent for this model instance
* notifyDestruction()
message coming back from instance agent that it has exited

- **putStatus(String)**
  alter the current status displayed for this model instance

- **setMark()**
  set the destruction mark

### Constructors

- **InstanceButton**

  ```java
  public InstanceButton(int n,
                      String name,
                      String status,
                      ClassAgent ownedBy)
  ```

  **Parameters:**
  
  - `n` - number identifying the remote model instance
  - `name` - name of remote model instance
  - `status` - description of current status of instance
  - `ownedBy` - ClassAgent in which this button appears

### Methods

- **putStatus**

  ```java
  public void putStatus(String status)
  ```

  alter the current status displayed for this model instance

- **makeAgent**

  ```java
  public synchronized void makeAgent()
  ```

  launch an instance agent for this model instance

- **getGraph**

  ```java
  public Object getGraph()
  ```

  **Returns:**
  
  the identity of the graphing applet

- **notifyDestruction**

  ```java
  public void notifyDestruction()
  ```

  message coming back from instance agent that it has exited

- **setMark**

  ```java
  public void setMark()
  ```
set the destruction mark

@clearMark

clearMark()

clear the destruction mark

@destroyed

destroyed()

message from class agent instructing a button and its agent to vanish if still marked for destruction

@action

action(Event event, Object obj)

handle presses of this button

Overrides:
action in class Component

@destroy

destroy()

actually destroys the button and any instance agent

@getAgent

getInstanceAgent()

Returns:
the instance agent associated with this button

@getNum

getNum()

Returns:
the number of the model instance

@getName

getName()

Returns:
the name of the model instance

@getStatus

getStatus()
Returns:

the current status of the model instance
public class ClassApplet
extends Applet
implements Runnable

Document: DRA/AS/SID/505/SD97006/1.0
Project: Generic Agents
Author: M J Corbin
Location: Systems Integration, DERA, Farnborough
Description: Presents a GUI Frame for launching a ClassAgent
Language: Java
Program File: ClassApplet.java
Issue History:
1.0 1/2/97 MJC First issue.

Variable Index
- tcp
  connection for sending commands to remote model class

Constructor Index
- ClassApplet()

Method Index
- action(Event, Object)
  handles action events associated with this applet
- createAgent(int, String, InstanceButton)
  creates a new instance agent and returns it
- destroy()
  destroys the ClassAgent associated with this applet
• init()
  applet interface for starting GUI
• notifyDestruction()
  close the connection to the remote model class
• run()
  entry for the unique thread associated with this applet
• start()
  called when window reappears
• stop()
  called when window is obscured

Variables

@ tcp

protected PrintStream tcp
  connection for sending commands to remote model class

Constructors

• ClassApplet
  public ClassApplet()

Methods

• init
  public void init()
    applet interface for starting GUI
    Overrides:
    init in class Applet

• start
  public void start()
    called when window reappears
    Overrides:
    start in class Applet

• stop
  public void stop()
    called when window is obscured
    Overrides:
stop in class Applet

*action*

```java
public boolean action(Event event, Object obj)
```

handles action events associated with this applet

**Overrides:**

```
action in class Component
```

*run*

```java
public void run()
```

entry for the unique thread associated with this applet

*destroy*

```java
public void destroy()
```

destroys the ClassAgent associated with this applet

**Overrides:**

```
destroy in class Applet
```

*createAgent*

```java
public InstanceAgent createAgent(int number, String name, InstanceButton button)
```

creates a new instance agent and returns it

**Parameters:**

- number - identifying number of the model instance
- name - name of the model instance
- button - the selector button for this model in the ClassAgent window

*notifyDestruction*

```java
public synchronized void notifyDestruction()
```

close the connection to the remote model class
Class TrackAgent

```java
public class TrackAgent
extends InstanceAgent
```

Document: DRA/AS/SID/505/SD97007/1.0
Project: Generic Agents
Author: M J Corbin
Location: Systems Integration, DERA, Farnborough
Description: Monitor and control a remote tracked vehicle model
Language: Java
Program File: TrackAgent.java
Issue History:
1.0 1/2/97 MJC First issue.

**Constructor Index**

- `TrackAgent(int, String, InstanceButton, PrintStream)`

**Method Index**

- `action(Event, Object)`
  handle control button actions
- `destroy()`
  remove the InstanceAgent and its windows
- `handleEvent(Event)`
  handle mouse inputs from the DragBox
- `readItemSpec(String, String, ItemPanel)`
  interpret the description for one model parameter
- `receiveData(String)`
  read in a string of data from the model
- `releaseControl()`
  release control of model instance
takeControl()
take control of model instance

**Constructors**

- **TrackAgent**
  
  public TrackAgent(int num,
                   String name,
                   InstanceButton button,
                   PrintStream request)

  **Parameters:**
  
  num - identifying number of the model instance
  name - name of the model instance
  button - the selector button for this mode in the ClassAgent window
  request - the stream through which the model can be addressed/controlled

**Methods**

- **receiveData**
  
  public synchronized void receiveData(String s)

  read in a string of data from the model
  
  **Overrides:**
  
  receiveData in class InstanceAgent

- **handleEvent**
  
  public boolean handleEvent(Event event)

  handle mouse inputs from the DragBox
  
  **Overrides:**
  
  handleEvent in class InstanceAgent

- **destroy**
  
  public synchronized void destroy()

  remove the InstanceAgent and its windows
  
  **Overrides:**
  
  destroy in class InstanceAgent

- **action**
  
  public boolean action(Event event,
                         Object obj)

  handle control button actions
  
  **Overrides:**
action in class InstanceAgent

- **takeControl**
  ```java
  public synchronized void takeControl()
  {
      take control of model instance
  }
  ```

- **releaseControl**
  ```java
  public synchronized void releaseControl()
  {
      release control of model instance
  }
  ```

- **readItemSpec**
  ```java
  protected void readItemSpec(String token, String s, ItemPanel panel)
  {
      interpret the description for one model parameter
      Parameters:
      token - string identifying the type of the parameter
      s - string containing the rest of the line
      panel - the panel which will contain the parameter’s fields
      Overrides:
      readItemSpec in class InstanceAgent
  }
  ```
Class TrackDisplay

```java
java.lang.Object
   \----- java.awt.Component
       \----- java.awt.Container
            \----- java.awt.Window
                 \----- java.awt.Frame
                    \----- TrackDisplay
```

class TrackDisplay
extends Frame

Document: DRA/AS/SID/505/SD97008/1.0
Project: Generic Agents
Author: M J Corbin
Location: Systems Integration, DERA, Farnborough
Description: Implements extra GUI frame for monitoring
and control of a remote tracked vehicle model
Language: Java
Program File: TrackAgent.java
Issue History:
1.0 1/2/97 MJC First issue.

**Constructor Index**

- TrackDisplay(TrackAgent)

**Method Index**

- `action(Event, Object)`
  pass all button actions back to the track agent
- `destroy()`
  hide this frame
- `handleEvent(Event)`
  handle a WINDOW_DESTROY event
- `readData(String)`
  interpret speed and heading values from remote model as "SPD value HDG value"

**Constructors**

```java
LXXV
```
**TrackDisplay**

public TrackDisplay(TrackAgent agent)

**Parameters:**
agent - TrackAgent which owns this GUI frame

**Methods**

**readData**

public void readData(String s)

interpret speed and heading values from remote model as "SPD value HDG value"

**handleEvent**

public boolean handleEvent(Event event)

handle a WINDOW_DESTROY event

**Overrides:**
handleEvent in class Component

**action**

public boolean action(Event event, Object obj)

pass all button actions back to the track agent

**Overrides:**
action in class Component

**destroy**

public void destroy()

hide this frame
Class StripDisplay

```java
java.lang.Object
 |--- java.awt.Component
 |    |--- java.awt.Canvas
 |    |--- StripDisplay
```

class StripDisplay
extends Canvas

**Document:** DRA/AS/SID/505/SD97009/1.0
**Project:** Generic Agents
**Author:** M J Corbin
**Location:** Systems Integration, DERA, Farnborough
**Description:** Implements a simple horizontal strip with scale and pointer
**Language:** Java
**Program File:** TrackAgent.java
**Issue History:**
1.0 1/2/97 MJC First issue.

---

**Variable Index**

- pixLength
  length of axis in pixels
- pixSize
  length of overall display area in pixels
- ticksCount
  number of ticks, inclusive of both end ones

---

**Constructor Index**

- StripDisplay(int, int, int)
  Horizontal strip display.

---

**Method Index**

- paint(Graphics)
  redraw the scale
- paintAxis(Graphics)
  override this to draw the axis in a different place
- paintPointer(Graphics)
draws a filled triangular pointer next to the axis
• **putData**(int, int, int, int)
  Positions the pointer and scales within the display area
• **scaleValue**(int)
  override this to alter how scale values are drawn
• **update**(Graphics)
  forces a paint

### Variables

- **pixLength**
  ```java
  protected int pixLength
  length of axis in pixels
  ```

- **pixSize**
  ```java
  protected int pixSize
  length of overall display area in pixels
  ```

- **ticksCount**
  ```java
  protected int ticksCount
  number of ticks, inclusive of both end ones
  ```

### Constructors

- **StripDisplay**
  ```java
  public StripDisplay(int length,
                      int numTicks,
                      int size)
  
  Horizontal strip display.
  **Parameters:**
  length - relates to the axis,
  numTicks - is inclusive of the ticks are both end of the axis
  size - relates to the overall display area.
  ```

### Methods

- **putData**
  ```java
  public void putData(int current,
                      int start,
                      int end,
                      int offset)
  ```
Positions the pointer and scales within the display area

**Parameters:**
- current - pointer value
- start - value at the beginning of the axis
- end - value at the end of the axis
- offset - pixel position of the start of the axis

```java
public void paint(Graphics g)
    // redraw the scale
    Overrides:
    paint in class Canvas
```

```java
protected void paintAxis(Graphics g)
    // override this to draw the axis in a different place
```

```java
protected String scaleValue(int f)
    // override this to alter how scale values are drawn
```

```java
protected void paintPointer(Graphics g)
    // draws a filled triangular pointer next to the axis
```

```java
public void update(Graphics g)
    // forces a paint
    Overrides:
    update in class Component
```
Class HeadingScale

document: DRA/AS/SID/505/SD970010/1.0
project: Generic Agents
author: M J Corbin
location: Systems Integration, DERA, Farnborough
description: Heading strip with moving tick marks, range 0 to 360
language: Java
program file: TrackAgent.java
issue history:
1.0 1/2/97 MJC First issue.

constructor index
• HeadingScale(int, int, int, int)

method index
• paintAxis(Graphics)
  draws the axis
• putHeading(double)
  insert a new heading value, in radians
• scaleValue(int)
  forces a wraparound of the scale values

constructors

• public HeadingScale(int length,
  int numTicks,
  int size,
  int interval)
Parameters:
length - relates to the axis,
numTicks - is inclusive of the ticks are both end of the axis
size - relates to the overall display area.
interval - heading interval between ticks, degrees

Methods

putHeading

public void putHeading(double heading)
insert a new heading value, in radians

paintAxis

protected void paintAxis(Graphics g)
draws the axis
Overrides:
paintAxis in class StripDisplay

scaleValue

protected String scaleValue(int f)
forces a wraparound of the scale values
Returns:
string value of wrapped scale value
Overrides:
scaleValue in class StripDisplay
Class DragBox

```java
java.lang.Object
  +---- java.awt.Component
        |    +---- java.awt.Canvas
        |    +---- DragBox
```

class DragBox
extends Canvas

**Document:** DRA/AS/SID/505/SD97011/1.0  
**Project:** Generic Agents  
**Author:** M J Corbin  
**Location:** Systems Integration, DERA, Farnborough  
**Description:** X and Y positioning box controlled by mouse  
**Language:** Java  
**Program File:** TrackAgent.java  
**Issue History:**  
1.0 1/2/97 MJC First issue.

**Constructor Index**

- DragBox(int, int, Component)

**Method Index**

- handleEvent(Event)  
  Handler for MOUSE_ENTER, MOUSE_EXIT, MOUSE_DRAG, MOUSE_UP events.  
- paint(Graphics)  
  redraw the box  
- update(Graphics)  
  force a paint

**Constructors**

- DragBox

  ```java
  public DragBox(int dx,  
                  int dy,  
                  Component owner)
  ```

  **Parameters:**
dx - width in pixels
dy - height in pixels
owner - containing window

**Methods**

@**handleEvent**

```java
global boolean handleEvent(Event event)
```

Handler for MOUSE_ENTER, MOUSE_EXIT, MOUSE_DRAG, MOUSE_UP events. A MOUSE_UP event is passed on to handleEvent of the owner

**Overrides:**
handleEvent in class Component

@**paint**

```java
global void paint(Graphics g)
```

redraw the box

**Overrides:**
paint in class Canvas

@**update**

```java
global void update(Graphics g)
```

force a paint

**Overrides:**
update in class Component
Appendix I  

Format of Communications between Model and Interaction Agent

Requests from agent to model are shown in plain text. Replies from model to agent are in italics. Brackets {} indicate repetition, while square brackets [] indicate optionality.

CLASS FILE

   CLASS FILE <file path>

   (used to indicate where to find the configuration file for this model)

CLASS INSTANCES

   CLASS INSTANCES [ { <id number> <name> <status keyword> } . . . ]

   (used to obtain a list of instances currently existing, and the status of each)

CLASS INSTANCES REPEAT <k>

   (repeat the reply above indefinitely, every k iterations, until cancelled)

CLASS INSTANCES CANCEL

   CLASS INSTANCES CANCEL

INST < id number > ALTER [ { <keyword> [ <value> ] } . . . ]

   INST < id number > ALTER ICS  
   (if the initial conditions were altered)

   INST < id number > ALTER CURRENT  
   (if the current values were altered)

   (used to control the values of parameters within the model)

INST < id number > PAGE <m>

   INST <id number> PAGE <m> <instance name> TIME <time tag>
      { <keyword> [ <value> ] } . . .

   (used to obtain one page-full of parameter values from the model)

INST < id number > PAGE <m> REPEAT <k>

   (repeat the page information above indefinitely, every k iterations, until cancelled)

INST < id number > PAGE <m> CANCEL

   INST < id number > PAGE <m> CANCEL
Appendix J  Format of Generic Agent Configuration File

STATUS <status keyword> <foreground colour> <background colour>

(used to specify the complete list of status keywords appliable to this model and to
associate each with a pair of display colours to attract the operator’s attention)

INSTANCE DATA

(used to signal the start of the per-instance parameters. Where no separate label is
provided for a parameter, the keyword is used as the label for the display)

INT | VINT ["<label>“] <keyword> [<units>]

(indicates an integer parameter. A VINT is a variable alterable by the operator)

FLOAT | VFLOAT ["<label>“] <keyword> [<units>] [<scaling value>]

(indicates a floating point parameter. VFLOATs are alterable)

ENUM | VENUM ["<label>“] <keyword> { <value> ... }

(indicates a set of enumeration values preceded by a keyword. VENUMs are alterable)

KEY | VKEY "<label>“ { <keyword> ... }

(indicates a set of mutually exclusive keyword values - the label is needed in this case.
VKEYs are alterable)