Autonomous agents for distributed virtual worlds

Wang, Fang

2006

Wang, F. (2006). Autonomous agents for distributed virtual worlds. Doctoral thesis,
Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/2518>

<https://doi.org/10.32657/10356/2518>

Nanyang Technological University

Downloaded on 26 Apr 2023 16:04:33 SGT

Nanyang Technological University

Autonomous Agents for Distributed Virtual Worlds

A thesis submitted to the Nanyang Technological University in fulfilment of
the requirement for the degree of Doctor of Philosophy

by

Wang Fang

Supervisor: Dr. Stephen John Turner

Division of Computer Science
School of Computer Engineering
Nanyang Technological University
Singapore 639798

September 2005

Abstract

Autonomous agents and multi-agent systems (MASs) provide a valuable tool for handling increasing software complexity and developing efficient applications. Large scale distributed simulation is a technology for linking simulation components at possibly geographically different locations to create a common virtual world. It is attractive to integrate these two technologies together, in order to facilitate rapid and accurate decision making in simulation applications, and to increase reusability and interoperability of agent-based simulation components. These are the reasons for our investigation of integrating agents into distributed simulations.

There are many applications such as battlefield simulations, interactive games, etc that provide the motivation for this integration. In this project, we develop a generic framework for autonomous agents in distributed simulations, where the behaviors of certain entities in the simulation are described by means of agents. The novelty of our approach comes from the way in which agents are integrated into a distributed simulation. We adopt the Foundation for Intelligent Physical Agents (FIPA) standard and choose a FIPA-compliant agent platform JADE to provide general agent-specific services. Components are linked using the Runtime Infrastructure (RTI) of the High Level Architecture (HLA), which is an IEEE standard for modeling and simulation. Thus we achieve a general integration architecture which is standard-compliant from the perspectives of not only agents but also distributed simulations.

To build a seamless integration of agents and distributed simulations, we develop a special kind of federate, called a gateway federate. Gateway federates enable agents to perceive information from the environment in which they are embedded and make responses to it. A prototype system has been implemented to demonstrate the feasibility of this approach. Agents are enabled to communicate with one another via the HLA RTI using a special kind of agent, called a mailbox agent, that ensures causality and

consistency of the message ordering. Communication is shown to be useful in achieving a better working efficiency for agents, overcoming the shortcoming of their limited knowledge about the environment.

When more and more agents are utilized in a distributed simulation, many kinds of agents may be involved and different levels of resolution may be required. The representation of agents at multiple levels of detail is also explored in this project. An approach using concurrent representations is adapted for agents, where the consistency is maintained using a dependency graph and mapping functions. The realization of this approach is obtained by constructing an architecture called a Cross-Resolution Agent Architecture (CRAA), whose flexibility is enhanced with the techniques of concurrent representations and aggregation-disaggregation. This agent architecture meets the requirements of linking agents of different resolution levels, permitting dynamic changes in resolution, and maintaining the consistency of representations.

Acknowledgements

Before the presentation of the research work, I need to thank many people who helped in this project and the development of my thesis.

First of all, I owe the greatest gratitude to my supervisor, Dr. Stephen J. Turner. He is such a nice man who always takes great pains giving me remarkable guidance. He has serious attitudes in research, and at the same time, abundant experiences in research areas. No doubt, I have learned many things and benefited a lot from him during these years.

Next, I would express my gratitude to my father and mother. In these years, they supported my work very much and gave me great understanding. They encouraged me to work hard through telephone lines, no matter if I have not much time to spend with them in China. And I want to thank Mr. Liu Song, who is also a research student at NTU. We discussed about resolving implementation problems sometimes, and shared some experiences and minds during the progress of our research work.

Finally, I need to thank Ms. Wang Lihua for her efforts in project discussion and assistance. And I wish to say thanks to Dr. Cai Wentong, Mr. Zeng Yi, Mr. Du Jiang, Dr. Zhou Suiping, and all the others who have provided much valuable advice and patient help to me. They all contributed to my research work in one way or another. Without all of their support, I could not have gone so far in my research work.

Contents

1	Introduction	1
1.1	Background	1
1.2	Motivation and Objectives	5
1.3	Organization of the Thesis	9
2	Agents	11
2.1	Agent Technology	11
2.1.1	Agents, Agent-Based Systems and MASs	11
2.1.2	BDI Agent Model	16
2.2	Agent Architectures	18
2.2.1	Fundamental Features of an Agent	18
2.2.2	A Basic MAS Architecture	20
2.3	Agent Toolkits	22
2.3.1	FIPA-Compliant Agent Toolkits	22
2.3.2	Agent Communication Languages	26
2.4	Summary	30

3	Distributed Simulation	31
3.1	Parallel and Distributed Simulation	31
3.1.1	Time in Simulations	32
3.1.2	Simulation Models	33
3.1.3	High Level Architecture	34
3.1.3.1	The HLA Standard	34
3.1.3.2	Time Management Services	36
3.1.3.3	Interest Management Services	38
3.2	Agents and Distributed Simulations	39
3.2.1	Integration Approaches	41
3.2.2	Communication Issues	44
3.3	Summary	46
4	Integration of Agents into Distributed Simulations	48
4.1	Agent Architecture	48
4.1.1	Sense	50
4.1.2	Deliberate	50
4.1.3	React	51
4.2	Overall Architecture	52
4.2.1	JADE	53
4.2.2	Gateway Federate	56
4.2.3	Simulation Cycle	58

4.3	Linking the Gateway Federate and Agent	60
4.3.1	Java RMI vs O2A Communication Channel	61
4.3.2	Synchronization using Latch	64
4.4	Summary	67
5	Mine Sweeping Game	68
5.1	Description of the Game	68
5.2	Soldier's Behaviors	71
5.2.1	Roam and Detect Mines	73
5.2.2	Pick Up Mine	76
5.2.3	Reach Border	78
5.2.4	Discard Mine	78
5.3	Implementation of the MSG	78
5.3.1	Federation Execution	80
5.3.2	Agent Behavior Scheduling	83
5.3.3	GUI of the MSG	86
5.4	Summary	86
6	Agent Communication in Distributed Simulations	88
6.1	Causality and Message Ordering	89
6.2	How Agents Communicate using HLA	90
6.2.1	Agent Communication Route	91
6.2.2	Requirements and Constraints	92

6.3	The Mailbox Agent	93
6.3.1	Message Transmission Procedure	95
6.3.2	Synchronization between Federates and Agents	96
6.4	Applying ACLs in the MSG	98
6.4.1	Communication among Soldiers	98
6.4.2	Message Templates	99
6.4.3	Freehands Hash Table	101
6.5	Experimental Results	102
6.5.1	Working Efficiency	103
6.5.2	Transmission Time of ACL Messages	105
6.6	Summary	109
7	Model Resolution in MAS Simulations	111
7.1	Descriptions of MAS Hierarchies	112
7.1.1	Horizontal Layered MAS	112
7.1.2	Vertical Layered MAS	114
7.1.3	Hybrid Layered MAS	116
7.2	Multi-Resolution Modeling	117
7.2.1	High and Low Resolution Models	118
7.2.2	MRM Approaches	119
7.3	Multi-Resolution Modeling in Agent Simulations	121
7.3.1	Concurrent Agent Representations	122

Ph.D Thesis

CONTENTS

7.3.2	Agent Resolution Transition	125
7.4	Maintaining Representation Consistency	127
7.4.1	Attribute Dependency Graph and Mapping Function	128
7.4.2	Consistency of Concurrent Agent Representations	129
7.4.3	Consistency of Agent Resolution Transition	130
7.4.4	Enforcing Consistency	132
7.5	Applying the ADG	133
7.6	Summary	138
8	Cross-Resolution Agent Architecture	140
8.1	Atomic Agent and Mass Agent	140
8.2	CRAA Algorithms	142
8.2.1	Two Agent Layers	144
8.2.2	Multiple Agent Layers	146
8.3	Applying the Mass Agent and CRAA	148
8.3.1	Mass Agent Architecture	148
8.3.2	Cross-Resolution Agent Architecture	151
8.3.3	Disaggregation Inconsistency Problem	154
8.4	Experimental Results	157
8.5	Summary	163
9	Conclusions	164
9.1	Achievements and Contributions	164

Ph.D Thesis

CONTENTS

9.2	Future Work	169
9.3	Summary	171

List of Figures

2.1	An agent	13
2.2	A BDI agent model	16
2.3	A basic MAS architecture	21
2.4	Agent management reference model in FIPA	24
2.5	Agent life-cycle as defined by FIPA	25
2.6	The syntax of a generic KQML message	27
2.7	The syntax of FIPA ACL messages	28
2.8	Agent identifier	28
3.1	Space-time diagram for (a) time-stepped and (b) event-driven simulations	34
3.2	A multi-agent environment using an agent middleware as proposed by Andersson and Löf	42
3.3	Middleware composed of JADE and gateway federates	44
3.4	Classification of communication	45
4.1	Our target agent architecture	49
4.2	Logical structure of an agent simulation cycle	49

4.3	An architecture for a MAS in an HLA-based simulation	53
4.4	JADE agent platform distributed over several containers	55
4.5	RTI ambassador and federate ambassador	56
4.6	Relations between a gateway federate and a federation	58
4.7	The simulation cycle of a federation	60
4.8	Two-JVM approach	62
4.9	One-JVM approach	62
4.10	Latencies of sending messages: (a) via JAVA RMI; (b) via JADE O2A communication channel.	64
4.11	Linking the agent and gateway federate: (a) the two-JVM approach; (b) use a shared latch object as the synchronization point in the one-JVM approach.	66
5.1	Framework of the MSG	70
5.2	Soldier behaviors in the MSG	72
5.3	Roam and detect mines behavior	74
5.4	Sensor range of soldiers	75
5.5	Agent direction choice	75
5.6	Soldier discovers a mine when moving to a new position	76
5.7	Pick up mine behavior	77
5.8	Reach border behavior	79
5.9	Discard mine behavior	79
5.10	Federation execution using sending interactions	80

5.11	Federation execution using DDM services	82
5.12	Enlarged subscription region of the soldier	83
5.13	Federation execution using DDM services with enlarged subscription region	84
5.14	Execution time of different versions	84
5.15	Flowchart of soldier behaviors	85
5.16	GUI of the MSG prototype	87
6.1	A scenario of distributed simulation leading to a violation of causality. . .	90
6.2	Agent communication route: (a) agents communicate directly via the communication channel provided by the agent platform (b) agents communicate indirectly via the RTI	92
6.3	Synchronization points	97
6.4	Flowchart of soldier behaviors with communication	100
6.5	Soldier behaviors in the MSG	101
6.6	Result with 1% mines distributed	103
6.7	Result with 10% mines distributed	104
6.8	Snapshots of the environment with, (a) distributed mines; (b) clustered mines.	105
6.9	Four agents for testing messages' transmission time	106
6.10	ACL messages' transmission time (one-way)	107
6.11	ACL messages' transmission time (two-way)	108
7.1	Horizontal layered MAS	113

Ph.D Thesis

LIST OF FIGURES

7.2	Vertical layered MAS	115
7.3	Hybrid layered MAS	116
7.4	Concurrent representations as proposed by Natrajan	120
7.5	Multi representation interaction as proposed by Natrajan	120
7.6	Concurrent agent representations	123
7.7	Agent aggregation	126
7.8	Agent disaggregation	127
7.9	A simple ADG	128
7.10	Concurrent agent representation consistency	130
7.11	Agent resolution transition consistency	131
7.12	Consistency enforcer	132
7.13	A commander-soldiers scenario	134
7.14	Dependency graph for the commander-soldiers scenario	137
7.15	Two soldiers and three mines	137
8.1	Peer agents	141
8.2	A mass agent representing three soldiers	143
8.3	Cross-resolution agent architecture	144
8.4	The mass agent for MSG	148
8.5	Three-phase behavior scheduling for soldiers	150
8.6	The CRAA agent for MSG	152
8.7	The CRAA agent for MSG before disaggregation	153

8.8	Disaggregation procedure	153
8.9	Disaggregation inconsistency	154
8.10	Dependency graph for the disaggregation phase	157
8.11	Federate configuration for comparing atomic agents, mass agents and CRAA agents	158
8.12	Execution time of atomic agents, mass agents and CRAA agents	159
8.13	Disaggregation time for atomic agents	160
8.14	Disaggregation time for a mass agent	161
8.15	Execution time of alternative solutions	162

List of Tables

2.1	Classification of some agent toolkits	26
2.2	FIPA ACL message parameters	29
6.1	Behavior of mailbox agent	94
7.1	The definitions of the commander's attributes	135
7.2	The definitions of soldiers' attributes	135
7.3	Mapping functions in the commander-soldiers scenario	136
7.4	Attribute values at time t	138
7.5	Attribute values at time t using consistency enforcer	138
8.1	The internal and external attributes determine the soldier's behaviors . .	150
8.2	Atomic agents, mass agents and CRAA agents	157

Chapter 1

Introduction

1.1 Background

Nowadays, software engineering develops so fast that almost every day dozens of software applications emerge. Today's software applications are mainly characterized by their component-based structures which are usually heterogeneous and distributed [16]. They need more functionality and a more friendly development environment, which means a more complicated underlying system is required. This in turn results in more complex code needing to be organized, managed and maintained. So the question is how the software developers can keep in pace with the fast changing market demands to develop a final efficient application.

Agent technology provides the right level of abstraction for developing this kind of application. Autonomous agents and multi-agent systems (MASs) provide a valuable tool for handling increasing software complexity and supporting efficient application development. They present a good way of analyzing, designing, and implementing complicated software applications [13, 29, 44].

Simply, an agent can be regarded as an encapsulated computer process that is situated in an environment and which is capable of flexible, autonomous action in that environment

in order to achieve its goals [33]. By autonomous, we mean that the agent has control over its own states and behaviors, where behavior is the agent's way of acting or conducting itself. An agent can receive inputs from the environment through its sensor and act on the environment through its effector. It may also communicate with other agents via some form of agent communication language (ACL)¹ and typically has the ability to engage in cooperative problem solving.

Research into agent-based systems, and in particular MASs, is leading the way in the development of techniques to address the variety of needs associated with these new technologies: languages and protocols which enable autonomous systems to communicate, frameworks for negotiation between self-interested parties, platforms which allow mobile processes to operate at remote sites, and many more [46].

A number of different approaches have emerged as candidates for the agent architecture, among which the belief-desire-intention (BDI) model [49] is probably the best-known, which views the agent as a rational entity having certain mental attitudes of *belief*, *desire* and *intention*. At the same time, dozens of environments for modeling, testing and finally implementing agent-based systems have been developed [65, 35, 39, 59]. An evaluation comparing many systems for developing software agents can be found in [16]. Jennings et al also provide a good overview of research and development in the field of autonomous agents and MASs [30]. They summarize some agent applications covering areas including industry, commerce, entertainment and medicine.

However, as mentioned in [46], early adopters of new technology tend to be wary when there are no commonly agreed, well-defined interfaces that are also backed by industry-supported standards. For this reason, and for the sake of supporting the development of various agent platforms and toolkits, reducing the work required to prepare for programming, and improving communication between different agents running on different

¹Communication among agents is supported by ACLs such as KQML [18], FIPA ACL [3] and KIF [27].

platforms, there has been a growing force behind developing agent standards, which directly results in the emergence of the FIPA standard [3] and the MASIF standard [2].

FIPA (the Foundation for Intelligent Physical Agents) is a non-profit organization established in 1996 to promote the development of specifications of generic agent technologies that maximize interoperability within and across agent-based applications. The FIPA agent standard provides specifications for an agent-enabling software framework. It aims to bring the commercial world a step closer to true software components, the benefits of which will include increased reuse, together with ease of upgrade. The FIPA agent standard was originally produced for stationary agents, while the OMG's MASIF (Mobile Agent System Interoperability Facility) regards the mobility of an agent from one location to another to be its defining characteristic. However, MASIF restricts the interoperability between agents to agents developed on a single type of platform whereas FIPA provides the freedom to develop agents on heterogeneous agent platforms, supporting interoperability directly. Although these two standards differ in whether they regard mobility to be an agent's inherent characteristic or not, they are starting to overlap and may ultimately converge [46]. There has also been strong interest within FIPA to extend its approach to support mobile agents.

Since the concept of agent has become widely used during the last few years, it has already become involved with simulations. Simulations are widely used today to analyze the behavior of physical systems or to create some computer-generated virtual worlds into which humans and/or physical devices are embedded. A *computer simulation* is defined as a computation that models the behavior of some real or imagined system over time. Basically every simulation specifies a physical system (or at least some of its components) in terms of a set of states and events.

Two kinds of simulations, namely *continuous simulation* and *discrete event simulation* are distinguished. A continuous simulation changes state continuously in time, while a

discrete event simulation fixes the occurrence of each event to a selected point in time which is instantaneous. Discrete event simulation can be used for analytic simulations and virtual environments (VEs), which form the two mainstreams of simulation applications historically [21]. Analytic simulation is the “classical” approach to simulation, which includes limited, or no, interaction with human participants or physical devices during the execution of the simulation program. A more recent approach is to create VEs. A principal goal in most VEs is concerned with achieving a “sufficiently realistic” representation of an actual or imagined system, as perceived by the human participants embedded into the environment, while what “sufficiently realistic” means depends on what one is trying to accomplish.

Large scale distributed simulation technology enables participants located in different geographical locations to share a common virtual world, which is called a Distributed Virtual Environment (DVE). Typical applications for this technology are training and entertainment including complex life-like or game-like simulated environments. There exist two main standards for building distributed simulation systems for DVE applications, namely, Distributed Interactive Simulation (DIS) [4] and its successor, the High Level Architecture (HLA) [5].

The HLA is an industry (IEEE-1516) standard for modeling and simulation. It can reduce the cost and development time of simulation systems and increase their capabilities by facilitating the reusability and interoperability of simulation components referred to as federates. It is increasingly being used in various simulation areas, including education, training, analysis, engineering, entertainment and games.

Commonly there are two main research areas in the cross field of combining agents with simulations, namely, (1) agent simulation: to simulate an agent system in order to learn more about its behaviors or to investigate the implications of alternative architectures [6, 38, 58, 60, 36], (2) agent-supported simulation: to utilize agents as a way of

controlling simulations and providing services [56, 26, 53].

In this thesis, the area of agent simulation also includes the area of agent-based simulation, where agents are used as entities in a simulation or virtual environment. Although agent simulation and agent-based simulation can be distinguished by the objectives of the simulation, the techniques used in these two areas are quite similar. So when one models a complex system using agents as entities (agent-based simulation), it is essentially also an agent simulation. Researchers can use the same model to study agent behaviors and architectures.

Parallel and distributed technologies enable a simulation program to be executed on a computing system containing multiple processors. The benefits of utilizing the HLA as a multi-agent environment are investigated in [6]. Most of the current research in using parallel and distributed simulation techniques in multi-agent and multi-agent based simulation is addressed to the first area above, namely, agent/agent-based simulation. Our work is also concerned with the first area, namely to develop autonomous agents for representing entities in distributed simulations. In the second area, agents may be used to provide some of the services of simulations such as performing data filtering in the HLA [56], or providing a voice-enabled interactive interface which is more natural to help users interact with each other and with the virtual world [53].

1.2 Motivation and Objectives

Rapid and accurate decision making is often critical to achieving success in many situations. For example, on modern battlefields, decision making is made more challenging by the rapid pace at which events occur and by the volume of information that can inundate the decision-maker. An approach to rapid and accurate decision making can be achieved by utilizing agent technology. At the same time, distributed simulation has become

increasingly important in recent years as a strategic technology for linking simulation components of various types at geographically different locations to create a common virtual world. So it is attractive to integrate agent technology and distributed simulation technology together to increase the speed and accuracy of decision making in distributed simulations, and on the other hand, the reusability and interoperability of agents. These are the reasons for our investigation of integrating agents into distributed simulations.

The novelty of our project comes from the way in which autonomous agents are integrated into a distributed simulation, with particular attention being given to the efficient use of HLA services [61]. There are many applications such as battlefield simulations, interactive games, etc that provide the motivation of this integration. The overall goal of this project is to develop a generic framework or architecture for integrating autonomous agents into distributed simulations. Here the objective is to develop a simulation application where the behaviors of certain entities in the simulation are described by means of agents. It may be an analytical simulation in which parallel and distributed techniques are used (e.g. an ecological simulation) or a DVE that might be used for training (e.g. a battlefield simulation). Such agents must be able to generate and monitor plans in an environment where only partial information is available and decisions must be made under time pressure. For example, in a battlefield simulation, an agent may need to make decisions such as to choose the best route to take [37] while uncertain about the exact position of the enemy. Agents should be proactive in being able to generate new plans in anticipation of future goals, as well as reactive, in monitoring and changing existing plans. They should also be able to pursue several goals at the same time, while responding appropriately to changes in their environment.

For building a seamless integration of agents and distributed simulations, a special kind of agent federate that supports agents is required. This kind of agent federate should enable agents to perceive information from the environment and make reactions

as responses to it. To meet different requirements of applications, we may need a single agent attached to a separate federate or several agents in a group attached to the same federate. Different federates may also need to be placed in the same machine or different machines. Although these agents are distributed, they must perceive a common view of the environment and their behaviors must be synchronized with each other.

To facilitate reusability and interoperability of simulation components (federates), a general integration architecture needs to be constructed. The architecture must be standard-compliant not only from the perspective of agents but also from the perspective of simulations. The execution of the agent federate should be compatible with standard agent architectures. The basic layers of a MAS including reactive layer, deliberative layer and social layer should be supported in the proposed architecture. Moreover, a developed agent federate must be reusable so that it can be embedded into other simulations as a component.

Since agent communication is regarded as one of the key characteristics of MASs, it is also a very important issue to be addressed in this project. A certain level of coordination provides a larger viewpoint for every individual agent which has only partial information about the environment. Communication overcomes the shortcoming of limited knowledge about the environment, and brings a better working efficiency for the agents compared to the situation where no agent communication is applied. However there is still the problem of how to enable the agent communication capabilities for a MAS that is integrated into an HLA-based distributed simulation. The focus is on keeping causality and consistency of the message ordering, which is crucial for communication especially in distributed simulations.

Multi-Resolution Modeling (MRM), which is a way of representing objects at multiple levels of detail, is also a very interesting issue that is explored in this project. In a distributed simulation where multiple agents are activated, various kinds of agents can

be involved, thus different levels of resolution may be required for those agents. At the same time, dynamic changes of the agents' model of existence are also necessary in some cases. For example in battlefields, one of the common soldier agents in a platoon may be promoted to a commander or a leader, or a battalion may break down into more and more detailed subdivisions hierarchically. Such cases are common in military simulations, and will result in a variation of resolutions. However, inconsistencies may arise during the time when agents are converted between different levels of resolutions. Thus a seamless design that permits changing resolution with the consistency of representation needs to be investigated.

An appropriate architecture which links existing agents with different resolutions should be constructed to meet these requirements. The proposed architecture involves both building agent hierarchies and supporting dynamic changes in representation resolution. In this way, users can build new models of agents or agent families so that they can easily change the resolution at which phenomena are treated. Both the techniques of aggregation-disaggregation and concurrent representations will be adopted in this architecture, and a tradeoff between these two solutions will be found. Finally, a general architecture is achieved, its flexibility being enhanced with these techniques.

In summary, the objectives of this project are as follows:

- To construct a general overall architecture for integrating agents into distributed simulations. In particular, the architecture should be compliant with agent and simulation standards, and it needs to provide a seamless integration of agents and distributed simulations.
- To facilitate agent communication in distributed simulations, which involves solving causality and consistency problems. Agent communication will be demonstrated to be effective in providing coordination among agents to overcome the shortcoming

of their limited view about the environment.

- To model resolution in MAS simulations. Issues of concurrent agent representations and agent resolution transition will be explored, and the problems about maintaining consistency should be resolved.
- To achieve a cross-resolution solution to enable concurrent agent representations and ensure consistency. Dynamic agent resolution transition and problems involved should also be addressed.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows: In Chapter 2, we briefly introduce related work concerning the research area of agents. Chapter 3 provides an overview of some basic techniques of parallel and distributed simulation, and a fundamental discussion of several research issues involved in combining agents and distributed simulation. Chapter 4 presents our approach to integrating agents into HLA-based simulations. Some synchronization problems are also investigated in this chapter. We have built a prototype system called MSG to lay the foundation for our investigations, which is illustrated in Chapter 5. Appropriate algorithms and methods are applied for synchronizing the execution of different federates and maintaining the consistency of the system. Chapter 6 mainly addresses the communication issue which is very important for MASs in distributed simulations. A mechanism enabling agent-to-agent communication is provided with the causality problem resolved. In Chapter 7, we present descriptions of MAS hierarchies and multi-resolution modeling in MAS simulations. Issues of concurrent agent representations and agent resolution transition are addressed, and problems about maintaining consistency are discussed. In Chapter 8, a cross resolution agent architecture is introduced, which involves building agent hierarchies and supporting dynamic changes in

representation resolution of MAS simulations. Finally, conclusions are given in Chapter 9.

Chapter 2

Agents

2.1 Agent Technology

Today's software applications are mainly characterized by their component-based structures which are usually heterogeneous and distributed [16]. Agent technology provides a method for handling increasing software complexity and supporting rapid and accurate decision making. A number of different approaches have emerged as candidates for agent architectures, and at the same time, dozens of environments for modeling, testing and finally implementing agent-based systems have been developed. We will review some related work about agents, agent architectures, agent toolkits, and agent communication languages in this section.

2.1.1 Agents, Agent-Based Systems and MASs

Autonomous agents and multi-agent systems (MASs) represent a new way of analyzing, designing, and implementing complicated software applications [30]. Jennings said that “the agent-based perspective offers a powerful repertoire of tools, techniques, and metaphors that have the potential to considerably improve the way in which people conceptualize and implement many types of software”. Agent technology has also been

demonstrated to be a useful tool for developing aspects of communications infrastructures [46]. Therefore, agents are more and more successfully deployed and used in practice. Also, MASs are being used increasingly in a wide range of application areas, including business process modeling, military simulations, resource allocation areas, advanced network control and Internet games.

Some key concepts in the field lack universally accepted definitions and there is no real agreement even on the core question of exactly what an agent is. Thus before we can continue our discussion, firstly we should make it clear what is meant by such terms as agent, agent-based system and MAS.

At the simplest, *agents* are independently executing programs which are capable of acting autonomously in the presence of expected and unexpected events [46]. The central concept that distinguishes software agents from simple programs is their interaction with their environment; typically, we say that an agent is *embedded* or *situated* within its environment [32].

Figure 2.1 shows a basic agent. An agent receives inputs from the environment through its sensor and acts on the environment through its effector. An agent may also communicate with other agents via some form of communication language [18] and has the ability to engage in cooperative problem solving.

Thus, we choose to use the definition given by Jennings and Wooldridge adapted from [64], which is a concise and precise description of an agent, that is:

An agent is an encapsulated computer process that is situated in some environment and is capable of flexible, autonomous action in that environment in order to achieve its goals.

There are three key concepts in this definition: *situatedness*, *autonomy*, and *flexibility* [30].

Figure 2.1: An agent

The notion of situatedness means that the agent lives in a real-world or software environment, receives sensory input from the environment in which it is embedded, and that it is able to perform actions which may change the environment in some way. The physical world or the Internet are both examples of such kinds of environments. This situatedness may be contrasted with the notion of disembodied intelligence that is often found in expert systems, because an expert system will not interact directly with any environment. The expert system receives information not via sensors, but through a user acting as a middleman, and in the same way, it does not act upon any environment.

Autonomy is not so easy to define precisely, but generally it means that the agent should be able to act without the direct intervention of human beings, and that it should have control over its own actions and internal states. Alternatively, in a stronger sense, some others use it to mean that the agent is capable of learning from experience [51]. However, in this thesis, by autonomous, we mean that the agent has control over its own states and behaviors without any assistance of human beings.

Although there are many examples of situated, autonomous computer software in existence, we still cannot consider them to be agents, this is because they are not capable of flexible action in order to meet their design objectives, which is the third concept in

the definition of an agent. Generally the term “flexible” means that the agent is [30]:

- *Responsive*: agents should perceive the environment in which they are embedded, and respond in a timely fashion to changes occurring in it.
- *Pro-active*: agents should not only simply act in response to their environment, but also be able to exhibit goal-directed behavior and take the initiative where appropriate.
- *Social*: agents should be able to interact with each other. If necessary, they communicate with humans or other agents in order to accomplish their goal or help others.

Wooldridge et al believe that these are the attributes that provide the power of the agent paradigm and distinguish agent systems from related software paradigms, such as object-oriented systems, distributed systems, and expert systems [63]. However some researchers may emphasize different aspects of agents from the above, such as *mobility* or *adaptability*. Of course, some special agents can have additional characteristics, and for certain types of applications, some attributes could be more important than others.

Now with the basic description of an agent in place, we continue to discuss about what an agent-based system is. Simply, an *agent-based system* refers to a system in which the key abstraction used is that of an agent. An agent-based system contains one or more agents. “In principle, an agent-based system might be conceptualized in terms of agents, but implemented without any software structures corresponding to agents at all” [30], as stated by Jennings and Wooldridge. It is possible to draw a parallel with object-oriented software, because it is entirely acceptable to design a system in terms of objects, but to implement it without the use of an object-oriented software environment. But this is unusual. A similar situation exists with agent technology. Thus generally an agent-based system is still expected to be both designed and implemented in terms of agents.

In agent-based systems, there are still cases in which a single agent solution is appropriate. This brings us to the notion of a Multi-Agent System (MAS), which is regarded as being more general and more interesting from a software engineering standpoint. A MAS is a system designed and implemented as several interacting agents. The first generation of MAS appeared in the 1980s, derived from *Distributed Artificial Intelligence* (DAI). From then on, the use of software agents has led to the development of new solutions and capabilities, especially in the area of network management and infrastructure.

Jennings et al have summarized some characteristics of MASs that distinguish them from traditional programs, which are the basic design and implementation requirements of a MAS [30]:

- *each agent has only partial information about the environment, i.e., a limited view-point;*
- *no global system control exists;*
- *data is decentralized;*
- *computation is asynchronous.*

MASs are claimed to be ideally suited to representing programs that have multiple problem solving methods, multiple perspectives and/or multiple problem entities. MASs are widely used in complex systems to study emergent behaviors. It is also well known that MASs not only have the traditional advantages of distributed and concurrent problem solving, but also the advantage of sophisticated patterns of interactions including *cooperation*, *coordination*, and *negotiation* [31]:

- *cooperation* : working together towards a common aim;
- *coordination* : organizing problem solving activity so that harmful interactions are avoided or beneficial interactions are exploited;

- *negotiation* : coming to an agreement which is acceptable to all the parties involved.

In summary, these different patterns of interactions and the characteristics illustrated above together distinguish MASs from other forms of software, and provide the underlying power of the paradigm.

2.1.2 BDI Agent Model

A number of different approaches have emerged as candidates for an agent model, among which the belief-desire-intention (BDI) model [49] is probably the best-known. Figure 2.2 shows the BDI agent model developed by Georgeff and Lansky [24]. The BDI model

Figure 2.2: A BDI agent model

combines psychologically based ideas, formal logic, architecture, implementations, and applications. It is a popular model for agent systems, in which *belief*, *desire*, and *intention* are identified as the three important elements in the problem-solving of an agent. In particular, we call the information on the state of the environment the agent's belief, which can be regarded as the agent's information component. It is also essential for the software agent to have information about its objectives, called, the agent's desire, which

can be thought of as representing the motivation state. Thirdly, an additional component represents the current chosen course of its actions. We call it the agent's intention. Here for intention, the predetermined sequences of actions or sub-goals that can accomplish specified tasks is referred to as its plan. In other words, intention captures the deliberative component of the agent.

In summary, the definitions of the three elements in the BDI model are as follows:

- **Belief** is an agent's local knowledge base that corresponds to information the agent has about the environment, which may be incomplete.
- **Desire** indicates what the agent is trying to achieve. It corresponds to the tasks allocated to it, and is sometimes called its *goal*.
- **Intention** is the agent's currently adopted plan or chosen desire, where plan is predetermined sequences of actions or sub-goals that can accomplish specified tasks. A set of plans forms the plan library of an agent.

When building complex distributed systems involving resource-bounded decision-making, the BDI approach appears particularly useful. Many systems have been implemented using the BDI model, including COSY [10], INTERRAP [42], and GRATE* [28], while the very first implementations were the Procedural Reasoning System (PRS) and Intelligent Resource-bounded Machine Intelligence (IRMA) [8], both developed for single-agent systems. From then on, many other implementations have emerged, especially adapting the model to meet the requirements of agents in MASs. Nowadays, the BDI model has come to be possibly the best known and best studied model of practical reasoning agents within the agent community [1]. Georgeff et al mentioned that perhaps the most compelling reason for BDI's success is that it combines a respectable philosophical model of human practical reasoning, a number of implementations, several successful

applications, including factory process control systems and business process management [23].

2.2 Agent Architectures

An *agent architecture* is a technological framework and scaffolding for developing agents. This section talks about different types of agent architectures. Several existing architectures of agent are presented, and characterized.

An agent architecture can be regarded as a methodology by which an agent can be constructed. However, in some cases, people narrow this definition to be closer to what seems to the more common usage of the term. For these cases, an agent architecture is a piece of software that allows the specification of an agent in an executable format. A complete architecture could be used to automatically generate a software agent from a specification. In any case, the output of an architecture is an agent.

Various kinds of agent architectures are applied in different applications. However, there exist some similarities among them. The features of agent architectures can be distilled through a process of reduction to a more standard architecture. For example, all agents need some basic components to complete their work, such as collecting information from the environment, and deliberating what to do in the next step. In this section, we will first introduce the fundamental features of a single agent. After that, the general layers of a MAS will be presented.

2.2.1 Fundamental Features of an Agent

Most agents have four components in their architecture: sensor, deliberation, effector and environment:

- (1) **Sensor.** The *sensor* component carries out the perception phase of an agent. A

sensor identifies the forms into which external information is translated so that it can be understood by an agent subsequently. However, a sensor's content depends on the dynamic conditions of the environment. In other words, the sensor determines what kind of external information could be delivered from the environment to the agent, but what the information will be still depends on the environment.

- (2) **Deliberation.** The *deliberation* component carries out the decision-making phase of a deliberative agent. As the deliberation is a time and space consuming process, in large simulations, it might be necessary to add a mechanism to pre-determine whether the agent should react or not. If the agent is not able to process the perception from the sensor at that time, this additional mechanism can release the agent from further deliberation and reaction.
- (3) **Effector.** The *effector* component carries out the reactive actions taken by an agent in response to the environment. In ideal cases, we suppose the information inside an effector is only updated if the agent goes through the deliberation phase and has some intentions to perform actions. Because communication is also a kind of action via which agents could affect the environment, we can devise two kinds of effectors according to whether or not they include agent-to-agent communication.
- (4) **Environment.** The *environment* where an agent is situated stands for all the external situations of the agent. Two kinds of environments can be distinguished, namely, the environment that also includes the physical world or virtual worlds and the environment that only consists of agents. In the former case, an agent's environment is composed of objects in the world. In the latter case, an agent's environment is composed of agents.

From the point of view of artificial intelligence, certainly an agent should behave like a human. A human being perceives the world by his/her sensors such as eyes and ears.

Obviously, the sensor's function is very similar to that of a person's sensor. The translation relating the external world to the internal understanding relies on specific sensors of agents. The form is determined by the sensor, while its content by the external world. A very simple example is that a person with good eyesight still cannot see anything in a black situation because nothing is visible there. So both the sensor and the environment influence the agent's perception phase. According to the agent architecture, each agent behavior is determined internally by the agent's deliberation process which makes decisions based on current perception. After that, external actions will be taken to alter the environment in response.

2.2.2 A Basic MAS Architecture

Basically, from the point of view of MASs, the most common structure of multiple deliberative agents consists of at least two layers: a *reactive layer* and a *deliberative layer* (see Figure 2.3). Each layer has its own model, which provides the representations for that layer. The reactive layer is related to the agents' perception and reaction components such as sensor and effector. It contains simple processes that generate sensory input from the environment to the agent, and the action output from the agent to the environment. For example, perceptual processing, including object identification and movement, need to be implemented at the reactive layer.

The deliberative layer contains the decision-making processes in which options are explicitly considered and evaluated before selection, such as knowledge-based planning, and behavior scheduling. This layer contains the agents' mental component such as deliberation. Sometimes it is also called the *mental layer*.

Besides these two main layers, many agents have another layer called a *cooperative layer*, especially in MASs which are much more complex than single agent systems. The cooperative layer sometimes may control the activities of the deliberative layer, providing

Figure 2.3: A basic MAS architecture

global monitoring functions. The cooperative layer is also called the *social layer*, and agent communication influences the operations of this layer.

The cooperative layer's function can be achieved by other methods, thus it is not necessary in each architecture. For example, the AMSIA agent architecture [55] uses a *blackboard* which enables collaboration between agents for their work processing. This approach can also be applied in the case of contracts among agents. In other words, every interaction is done in an indirect way through the environment, and there is no explicit coordination. In this situation, agents choose an action depending on their control algorithm and local perception, they do not explicitly negotiate, nor explicitly identify and handle antagonism situations.

Further discussions about the the reactive layer and the deliberative layer can be found in Chapter 4, and the capabilities of the social layer will be investigated in Chapter 6. We need a general architecture in which the model of an agent can be more widely used in different domains. That is the reason why we investigate similarities and differences among existing agent architectures to contribute some ideas for developing a general agent architecture.

2.3 Agent Toolkits

A large number of existing general-purpose environments for modeling and developing MASs have been built [65, 35, 39, 59], which we refer to as agent toolkits in this thesis. Agent toolkits are defined as sets of components from which to build agent platforms (APs) and sets of tools to help operate agent platforms. The concept of agent toolkits emerged in the second generation of agent-based systems in the 1990s. The agent-based system is different from the agent toolkit itself in that when it executes, it represents one or more particular configuration(s) of the toolkit.

2.3.1 FIPA-Compliant Agent Toolkits

Many agent toolkits which range from open source to commercial have been made available for use by third-party developers, application developers and end-users. In these toolkits, common functions such as agent-level communication, message transport and a directory service were encapsulated into components with well-defined interfaces using mainstream languages such as C++ and, more commonly, Java. In a heterogeneous world, concurrent distributed development has led to many types of MASs that are islands of functionality, namely, agents on different types of platform are unable to interoperate with each other. Agents from different vendors are also likely to use different types of message formats and the meaning, and interpretation of the content is likely to differ. Accordingly, for the sake of supporting the development of various agent platforms, reducing the work required to prepare for programming, and improving communication between different agents running on different platforms, there is a great need to develop standards. This resulted in the establishment of the FIPA [3] in 1996.

FIPA is one of the most well-known organizations in developing agent standards, with the role of producing software standards for heterogeneous and interacting agents and

agent-based systems. FIPA produced its first set of seven specifications in 1997. This set of specifications includes an agent architecture referred to as the FIPA 1997 agent platform, an agent communication language, and some applications such as travel assistance, network management, audio-visual entertainment and personal assistance. In subsequent years, further specifications were added, the architecture and agent communication language have been refined, the specification process has been formalized and there is more focus on abstractions and instantiations of the abstractions for a heterogeneous world.

Currently, there are several versions of the specifications for the FIPA standard in existence, while their two core parts are:

- Part one, which defines the agent platform or *agent management reference model* (see Figure 2.4);
- Part two, which specifies the FIPA ACL.

The agent management reference model provides the normative framework within which FIPA agents live and execute. It consists of an *agent management system* (AMS), a *message transport system* (MTS) and a *directory facilitator* (DF), which are the three mandatory roles identified in an agent platform. Combined with the agent life cycle as shown in Figure 2.5, it establishes the logical contexts for the creation, operation and retirement of agents. The AMS and MTS control inter-agent communication. The MTS supports interoperability both within and across agent platforms. The DF provides a “yellow pages” service to other agents. They are the specific capability sets for a FIPA-compliant agent platform to support agent management.

Table 2.1 which is adapted from [46] classifies some current well-known agent toolkits according to the standards they support at the agent level, and highlights the languages in which they are implemented. In addition, it indicates whether they support stationary agents or mobile agents, or both. Among them, JATLite was the first stationary agent

Figure 2.4: Agent management reference model in FIPA

toolkit made publicly available; FIPA-OS was the first one to support the FIPA standard and was made available under full open source, released in October 1999. Since that time, other agent toolkits such as BT's ZEUS platform [9], and TILab's JADE [7] have emerged.

Poslad et al mention that most agent platforms, including FIPA compliant and non-FIPA compliant platforms, naturally offer openness at the agent level; whether or not the platform itself is fixed or closed, service and user agents can be dynamically added to the platform and can interoperate [46]. This requires a common means of representing (encoding), exchanging (protocol), and understanding (ontology) service information.

Figure 2.5: Agent life-cycle as defined by FIPA

Sometimes some common understanding of the meaning of the messages is required, which is a specialized knowledge component called an *ontology* [25]. The ontology can specify the objects, concepts and relationships in a given domain.

FIPA compliant platforms define a standard protocol based on speech act theory [52] and several standard ontologies: a core one for registering, querying and deregistering services (part of the FIPA management ontology) and various domain-specific ones. There is no mandatory specific service encoding for FIPA compliant platforms. Non-FIPA compliant platforms have no common means of encoding, exchanging and understanding. They require the use of a combination of platform-specific encoding, protocol and service ontology to represent, exchange and understand service information.

Agent Toolkit	Mobile	Stationary	FIPA standard	Other Standards	ACL	Language
AgentBuilder	yes	yes	no			JAVA
Aglets	yes	no	no	MASIF		JAVA
Comtec Agent Platform	no	yes	yes		FIPA ACL	JAVA
Concordia	yes	no	no			JAVA
DMARS	no	yes	no			C/C++
FIPA-OS	no	yes	yes		FIPA ACL	JAVA
Grasshopper	yes	no	yes	MASIF	FIPA ACL	JAVA
Jackal	no	yes	no		KQML	JAVA
JADE	yes	yes	yes		FIPA ACL	JAVA
JAFMAS	no	yes	no		KQML	JAVA
JATLite	no	yes	no		KQML	JAVA
MOLE	yes	no	no			JAVA
Open Agent Architecture	no	yes	no		KQML	JAVA,C,VB,Lisp,Prolog,etc.
Voyager	yes	no	no			JAVA
ZEUS	no	yes	yes		FIPA ACL	JAVA

Table 2.1: Classification of some agent toolkits

2.3.2 Agent Communication Languages

In a MAS, since each agent has only partial information about the environment, agent communication is required in order to overcome the shortcoming of limited knowledge of the agents, and brings a better working efficiency for them. In this case, agent developers may need to select an appropriate language to support the communication among agents. According to [46], any act of communication involves three aspects:

- the method of message passing,
- the format, or syntax, of the information being transferred, and
- the meaning, or semantics, of the information (and message).

Agent communication has been successfully modelled based on speech act theory [52]. In speech act theory, a term *performative* is used to identify the interaction behind a spoken communication, including verbs such as *request*, *tell*, *report*, *commit* and *reply*.

```

(KQML-performative
  : sender <word>
  : receiver <word>
  : language <word>
  : ontology <word>
  : content <expression>
  ...
)
```

Figure 2.6: The syntax of a generic KQML message

These performatives can be used to constrain the semantics of messages among agents, and to simplify how agents should react to the messages they receive.

Knowledge Query and Manipulation Language (KQML) [17] is one of the first initiatives to specify how to support the social interaction characteristics of agents using a protocol of speech act theory. The syntax of a generic message in KQML is given in Figure 2.6. However this language is not a standard one that has a single specification (or set of specifications) ratified by common agreement of organizations or communities. Variations of KQML exist and different agent systems may use different ones which prevent full interoperability between them. Other than the lack of an agreed specification, KQML has another shortcoming of having no well-defined semantics. Using performatives alone is insufficient to guarantee that messages will be interpreted and acted on correctly by other agents.

The emergence of the FIPA specification of its own agent communication language provides a standard way of processing messages. In this way, a FIPA-compliant agent could easily make it clear to other FIPA-compliant agents what the purpose of its messages are.

The FIPA ACL defines what is considered to be parameters corresponding to performatives for agent communication (see Table 2.2). It provides a flexible approach for com-

munication between software agents. A universal message-based language approach has been made providing a consistent speech-act-based interface, and asynchronous message-based interaction between agents is enabled. The syntax of FIPA ACL is shown in Figure 2.7, and the syntax of an agent identifier is given in Figure 2.8.

```
(MessageType
  : sender <agent identifier>
  : receiver <agent identifier set>
  : content <string>
  : reply-with <expression>
  : reply-by <date time>
  : in-reply-to <agent identifier>
  : reply-to <agent identifier set>
  : language <expression>
  : encoding <expression>
  : ontology <expression>
  : protocol <word>
  : conversation-id <expression>
  : X-<user defined parameter> <expression>
  ...
)
```

Figure 2.7: The syntax of FIPA ACL messages

```
(AgentIdentifier
  : name <word>
  : addresses <URL sequence>
  : resolvers <agent identifier sequence>
  : X-<user defined parameter> <expression>
)
```

Figure 2.8: Agent identifier

We can see that to control the conversation, the FIPA ACL defines two additional fields *reply-with* and *in-reply-to*. The *conversation-id* field is also defined as an aid to the conversation management. Not all of the parameters in Table 2.2 need to be used by each agent, and different types of combinations are allowed. However this flexibility makes it

Parameter	Category of Parameter	Description
performative	Type of communicative act	Denotes the type of the communicative act of the ACL message
sender	Participant in communication	Denotes the identity of the sender of the message.
receiver	Participant in communication	Denotes the identity of the intended recipients of the message
reply-to	Participant in communication	Indicates that subsequent messages in this conversation thread are to be directed to the agent named in the <i>reply-to</i> , instead of the agent named in the <i>sender</i>
content	Content of message	Denotes the content of the message; equivalently denotes the object of the action
language	Description of Content	Denotes the language in which the <i>content</i> parameter is expressed
encoding	Description of Content	Denotes the specific encoding of the content language expression
ontology	Description of Content	Denotes the ontology(s) used to give a meaning to the symbols in the content expression
protocol	Control of conversation	Denotes the interaction protocol that the <i>sender</i> is employing
conversation-id	Control of conversation	Introduces an expression (a conversation identifier) which is used to identify the ongoing sequence of communicative acts that together form a conversation
reply-with	Control of conversation	Introduces an expression that will be used by the responding agent to identify this message
in-reply-to	Control of conversation	Denotes an expression that references an earlier action to which this message is a reply
reply-by	Control of conversation	Denotes a time and/or date expression which indicates the latest time by which the <i>sender</i> would like to receive a reply

Table 2.2: FIPA ACL message parameters

possible that in a heterogeneous world, different agents may encode and transport the content differently.

Looking through current agent platforms, we can find that most of them are using these two kinds of ACLs, namely KQML and FIPA ACL. However, even if agents share a common language, they may not understand each other. Only the agents that share a common language and an ontology can communicate effectively both syntactically and semantically [46].

2.4 Summary

In this chapter, we have summarized some current technologies in the area of agents. Agent technology provides a valuable tool for handling increasing software complexity and supporting the development of efficient applications. We have addressed the issues of how to construct a general architecture for a single agent or a MAS. In the following chapters, we will develop autonomous agents for representing entities in distributed simulations and present a general architecture for this proposal.

Chapter 3

Distributed Simulation

In this chapter, we first provide an overview of some basic techniques of parallel and distributed simulation. We then discuss some fundamental research issues involved in this project. These include how to construct an overall architecture for integrating a FIPA-compliant agent platform into an HLA-based simulation, and important communication issues to be investigated in the following chapters. This chapter can be regarded as a preparation for our further investigation.

3.1 Parallel and Distributed Simulation

As described in Section 1.1, simulations are basically classified into two categories: continuous simulation and discrete event simulation. From now on, when talking about simulations, we only address discrete event simulation which is more widely-used in practice. Especially, by the notion of *parallel and distributed simulation* (PADS), we refer to a simulation that is executed on parallel/distributed computer systems, namely a system composed of multiple interconnected computers.

There are primarily four principal benefits of executing a simulation application across multiple computers: reducing execution time, enabling geographical distribution, integrating simulation components that execute on machines from different manufacturers,

and supporting fault tolerance [21]. In this section, we provide an overview of the technologies in this area as a basis for our research work and further investigation.

3.1.1 Time in Simulations

Time is a very important concept in the area of simulation. Before further discussion, we need to explain several related concepts and make it clear what is meant by the notions of *physical time*, *simulation time*, *wallclock time* and *real-time* [21]:

- *Physical time* refers to time in a physical system.
- *Simulation time* is the simulation's representation of physical time in the system being modelled. It is also referred as *virtual time* (VT) in this thesis.
- *Wallclock time* refers to time during the execution of a simulation program. A simulation program can usually obtain the current value of wallclock time by reading a hardware clock maintained by the operating system.

Simulation time or VT is a concept that only exists in simulated worlds. Simulation executions where advances in simulation time are paced by wallclock time are called *real-time executions*, and simulation applications designed to operate in this mode are *real-time simulations*.

In a real-time simulation, although there exists a relationship between simulation time and wallclock time, these two concepts need to be kept distinct. Simulation time may advance faster or slower than wallclock time by some scale factor. This kind of simulation is called *scaled real-time simulation*. For example, if the scale factor is 2, the simulation runs twice as fast as wallclock time. If the scale factor is less than 1, the simulation runs slower than wallclock time. A real-time simulation is a special case of scaled real-time execution where the scale factor is set to 1. Both real-time and scaled

real-time simulations use a mapping function to convert wallclock time to simulation time, $T_s = W2S(T_w) = T_{Start} + Scale * (T_w - T_{wStart})$, in which T_s is the simulation time, T_w is the wallclock time, T_{Start} is the simulation time at the beginning of the simulation, $Scale$ is the scale factor, and T_{wStart} is the wallclock time at beginning of the simulation.

3.1.2 Simulation Models

A *simulation model* is the specification of a physical system (or at least some of its components) in terms of a set of *states* and *events*¹. Considering the mechanism used by the simulation to advance simulation time, we can distinguish the two most common types of simulations called *time-stepped* and *event-driven* simulations [21], as shown in Figure 3.1:

- *Time-stepped simulation*: simulation time is subdivided as a sequence of equal-sized time-steps (or ticks), and the simulation advances from one time-step to the next. In other words, the observation of the simulated dynamic system is dispersed by constant time intervals Δ .
- *Event-driven simulation*: the simulation time is advanced to the time of the next event where each event has a time-stamp associated with it that indicates the point in simulation time when the event occurs. In other words, the observation of the simulated system is dispersed at event occurrence instants.

In time-stepped simulations, the choice of time interval Δ interchanges simulation accuracy and elapsed execution time: ticks short enough to guarantee the required precision generally imply longer execution time. For event structures irregularly dispersed over time, the time-stepped concept leads to inefficient simulation executions. Rather

¹An event is an abstraction used in the simulation to model some instantaneous action in the physical system. Each event usually results in some change in one or more state variables defined by the simulation.

Figure 3.1: Space-time diagram for (a) time-stepped and (b) event-driven simulations

than compute new values for variables at each time-step, it may be more efficient to only update the variables when an event occurs. This is the key idea behind event-driven simulation.

3.1.3 High Level Architecture

The High Level Architecture (HLA) was originally a Defense Modeling and Simulation Office (DMSO) standard for simulation software. However, it has already become an industry (IEEE-1516) standard [5] that is designed to promote the reusability and interoperability of simulation components.

3.1.3.1 The HLA Standard

In the HLA, a distributed simulation is called a *federation*, and each individual simulation component is referred to as a *federate*, one point of attachment to the *RunTime Infrastructure* (RTI) which is an implementation of the HLA standard. An individual component can be a computer simulation, it can also be a physical device, a passive data viewer or an interface to a human participant.

Basically, the HLA standard comprises three major components including:

1. HLA rules,
2. HLA interface specification, and
3. HLA Object Model Template (OMT).

The latest version of standard also includes Federation Development and Execution Process (FEDEP) Model which provides a general-purpose framework for identification and description of special actions required in the particular domain of interest.

The HLA rules define the responsibilities of federates and their relationships with the federation. There are two sets of rules including rules applied to the federation and rules applied to the federates. We must apply these rules and solve any problems that will occur when integrating agents into distributed simulations using the HLA.

The interface specification as implemented by the RTI defines how federates interact with the RTI. The RTI provides services in a manner that is comparable to the way a distributed operating system provides services to applications. The interface specification also identifies *callback* functions each federate must provide.

Using the OMT, a common method for recording information is provided. Three formats of key models are established by the OMT, including Federation Object Model (FOM), Simulation Object Model (SOM) and Management Object Model (MOM). The FOM defines the overall data to be exchanged between federates during the simulation execution including objects and interactions. The SOM focuses on the data each federate is willing to exchange, while the MOM is designed to provide insight into the operations of federations. The MOM identifies object and interactions used to manage a federation.

The facilities provided by the RTI are classified into six categories: federation management, declaration management, object management, ownership management, time

management and data distribution management [21]:

- *Federation Management*: allows federates to create and destroy the federation execution, and join or resign from an existing federation.
- *Declaration Management*: allows federates to establish their intent to publish object attributes and interactions, and to subscribe to updates and interactions produced by other federates.
- *Object Management*: allows federates to create and delete object instances, and produce and receive individual attribute updates and interactions.
- *Ownership Management*: allows federates to transfer the ownership of object attributes during the federation execution.
- *Time Management*: coordinates the advancement of simulation time of federates, and (if appropriate) its relationship to wallclock time.
- *Data Distribution Management (DDM)*: reduces unnecessary information to be transferred between federates by filtering out irrelevant data.

3.1.3.2 Time Management Services

In the HLA, the principal types of time management mechanisms used for simulations have been described in [20]: event-driven, time-stepped, parallel discrete-event simulation and wallclock-time driven. These cover a broad spectrum of applications and simulations. The requirements in developing these applications and simulation also drive the design of the HLA time management services.

In the HLA federation, a *time-stamp order* (TSO) event is an event with an associated time-stamp. A federate may be “regulating”, “constrained”, “regulating and

constrained” or “neither regulating nor constrained”. A *regulating federate* is a federate that is capable of generating TSO events. A regulating federate promises that any TSO events it generates will occur no earlier than $t_{current} + t_{lookahead}$, where the $t_{lookahead}$ represents a contract between the regulating federate and the federation. Each federate using the time management services must declare a lookahead. A federate may declare its lookahead to be zero, in which case it can generate events at the current simulation time. However, this may result in poor performance.

Regulating federates regulate the progress in time of federates that are designated as *constrained*. A *constrained federate* is a federate that is capable of receiving TSO events. A constrained federate cannot proceed beyond its current LBTS (Lower Bound Time Stamp) that specifies the time of the earliest possible TSO event the federate can receive. All federates have an LBTS value, but it is only meaningful to constrained federates or unconstrained federates planning to become constrained. Constrained federates must request time advance through the RTI before they can advance their local time, which are called *time advance requests* (TARs) for time-stepped simulations and *next event requests* (NERs) for event-driven simulations. A time advancement request is granted only if no events containing a smaller time-stamp will later arrive at this federate. Unconstrained federates are free to progress through time. They have no requirement to request time advance through the RTI.

In the HLA/RTI simulation model, there will be no universal time. Each federate is allowed to use its own time management policy, and at any given point each federate could have different current times. In our project, we use the TSO event delivery service of the HLA, whose properties are listed below:

1. The sender must be time-regulating.
2. The receiver must be time-constrained.

3. The event is identified as TSO in the FOM.
4. The time-stamp on the event must be at least $t_{current} + t_{lookahead}$.

Furthermore, in a time-stepped simulation, the local virtual time t advances with a fixed time increment Δt . In this way, all of the federates in a federation advance their time together.

3.1.3.3 Interest Management Services

Interest management (IM) is used in distributed simulations to reduce communication requirements. IM ensures the simulated entities only receive information they need during the simulation execution. To use IM, entities have to declare their preferences of specific data first, then the infrastructure will match their interests and the data needed will be transmitted between matching entities. Various IM schemes have been devised over the years, utilizing different communication models and filtering methods [41]. In many existing systems, IM is realized via the use of IP multicast addressing, where data is sent to a selected subnet of all potential receivers.

The IM services specified for the HLA are the latest in a succession of mechanisms for large scale distributed simulations. The HLA supports two types of filtering including:

- *Class-based filtering.* This uses Declaration Management (DM) services. DM services allow a federate to update and receive updates to object attributes based solely on object class.
- *Value-based filtering.* This uses Data Distribution Management (DDM) services. DDM services extend the DM services using routing spaces and regions.

The fundamental concept used in the HLA to support value-based DDM is the routing space. A routing space is a normalized multidimensional coordinate system, where

federates indicate interests in receiving or providing updates via subscription regions or update regions. By calculating the intersection of subscription and update regions, the RTI establishes connectivity and thus provides efficient data transfer in the federation. The DMSO RTI version 1.3 provides a grid based approach, which statically breaks up a routing space into grid cells, and assigns a channel to each cell [5]. The subscription and update regions are mapped into the grid cells. An update is sent out on all channels corresponding to cells that overlap the update region. Entities subscribe to the channels matching the cells that overlap the subscription region. As entities change interests, they change channels correspondingly by notifying the RTI about the region modification.

3.2 Agents and Distributed Simulations

In this section, a brief overview of the main research areas in combining agents with distributed simulations will be presented. Firstly let us recall the two main research areas that have been identified in Section 1.1:

- **Agent simulation:** to simulate an agent system in order to learn more about its behaviors or to investigate the implications of alternative architectures.
- **Agent-supported simulation:** to utilize agents as a way of controlling simulations and providing services.

In the first area of agent/agent-based simulation, agents make it possible for distributed simulations to achieve rapid and accurate decision making. An application may require different kinds of agents, with different properties. Here, the HLA can be used to support component-based development of simulation models. This is even more attractive because multi-agent systems are inherently distributed and structured. Moreover, “situated” agents may need to interact with another existing simulation component which

is not an agent. In this case, the agent federates and the environment in which they are situated can be linked easily using the RTI. In other words, combining MAS technology and the HLA together provides distributed simulations with intelligence and decreases the complexity of simulation development. In agent simulation, the HLA not only supports the reusability of MAS components, but also facilitates their interoperability.

Andersson and Löf [6] presented a way of using the HLA/RTI as a multi-agent environment to develop an overall architecture for agent behavior. Some of the benefits of using the HLA as a conceptual basis for a multi-agent environment were investigated and an air combat scenario was developed. They have extended the HLA/RTI with KQML [17], an ACL.

Logan and Theodoropoulos [38] discussed the application of distributed techniques to the simulation of multi-agent systems, where the efficient distribution of the agent environment is identified as a key problem in the simulation of agent-based systems. They have also extended the SIM_AGENT toolkit [54] with the HLA to enable it to be distributed [36]. Minson and Theodoropoulos also discussed the design and implementation of a system that permits integration, through an HLA federation, of multiple instances of the Java-based lightweight-agent simulation toolkit RePast [40].

Moreover, taking the dynamic pattern of composition and interaction of agents into account, Uhrmacher and Gugler claimed that parallel and distributed simulations are usually required for testing agents [57]. Most current test-beds for MASs do not execute their models concurrently. In the case where a large number of agents need to be tested, which consumes significant space and computation resources, a concurrent, distributed simulation layer is necessary. Uhrmacher and Gugler have developed JAMES (a Java-based agent modeling environment for simulation) to meet this requirement [58].

In the second area of agent-supported simulation, agents can be used to provide some of the services of simulations. Tan and Xu have employed intelligent agents to perform

data filtering in the HLA [56]. They have implemented an agent-based DDM (Data Distribution Management) in the RTI, and have compared this approach with two other general filtering mechanisms including region-based and grid-based DDM. Two scenarios are used to analyze experimental results, an AWACS sensing aircraft simulation and an air traffic control simulation. Besides this work, an agent-aided collaborative virtual environment (CVE) over the HLA/RTI has been investigated. Shen et al have described a CVE system over the Internet in which voice-enabled interactive agents provide a more natural interface to help users interact with each other and with the virtual world [53].

3.2.1 Integration Approaches

In our project, a general overall architecture is developed for integrating agents into distributed simulations. The architecture should have great flexibility to be applied in many applications, and the simulation components in an application using this architecture should be reusable in many other simulations. There are different approaches that may be adopted to construct the overall architecture for integrating agents into an HLA-based simulation. Generally, a fundamental concern is to construct a feasible middleware between the agents and the RTI.

Andersson and Löf give an early proposal for the use of the HLA/RTI as a platform for a multi-agent environment. They extend the HLA/RTI with an agent communication language, such as KQML, to provide the features for supporting agent communication [6]. Thus in the implementation, they add a KQML-layer to every federate that hosts an agent, in order to make the agents successfully communicate with each other. They develop an air combat scenario to test the environment which can host agents and support communication and information distribution. Moreover, a conclusion is drawn that if the HLA/RTI is extended with an agent-specific services middleware, it would provide a suitable environment for intelligent agents. Figure 3.2 shows their integration proposal.

However no further references for this proposal have been found in the literature to demonstrate its feasibility or to evaluate the performance.

Figure 3.2: A multi-agent environment using an agent middleware as proposed by Andersson and Löf

Lees et al also present an approach to distributed simulations of agents using the SIM_AGENT toolkit and the HLA interoperability framework [36]. They outline the changes necessary to the SIM_AGENT toolkit to allow integration with the HLA, and compare the performance of the distributed SIM_AGENT version with the original non-distributed one. A simple Tileworld scenario is used as an example to show how the HLA can be used to flexibly distribute a SIM_AGENT simulation with different agents simulated on different machines. They briefly describe the simulation cycle of the combined system which is called HLA_AGENT. The integration is transparent in the sense that the existing SIM_AGENT code runs unmodified and the agents are unaware that other parts of the simulation are running remotely.

Andersson et al's proposal can be regarded as the origin of our investigation of integration architectures, but their requirement is not quite the same with ours. What we want to achieve is not simply to provide a communication language for the agents in DVEs, but a general integration architecture that could be suitable for many applications. In

their proposal, non-agent federates access the RTI directly, while agent federates do so indirectly via the agent middleware. The agent federates are above the layer of agent-specific services. In our proposal, above the RTI, a FIPA-compliant agent platform (AP) is utilized, as part of the overall architecture, providing general agent-specific services.

Lees et al's preliminary experiences are also valuable for our investigation of integration. However, their architecture is based on a specific toolkit, SIM_AGENT, which is not widely used. This is why we integrate a standard agent toolkit with a HLA-based simulation instead of a toolkit such as SIM_AGENT. In our proposal, with the standard RTI infrastructure in place at the low level, a similar standard agent platform will be used at the high level. We select a FIPA-compliant agent toolkit such as JADE (Java Agent Development Framework) [7] to provide the agent-specific services and to support a standard ACL for the agents. In this way, a general overall architecture is achieved, which is reusable and has flexibility to be applied in many other simulations.

In our approach, for the seamless integration of standard agents with the HLA RTI, a special kind of federate, called a *gateway federate*, is constructed between them. The gateway federate can take charge of agents, and synchronize their activities with other federates with or without agents. The gateway federates still access the RTI directly. Agent containers where agents reside are constructed upon them. As shown in Figure 3.3, we propose a *middleware* composed of the standard agent platform and the gateway federate in this approach. In summary, Andersson's proposal has been advanced in an alternative way, and at the same time, compared with Lees et al's approach, the changes that might be necessary to the agent toolkit are reduced. Finally, we obtain a more general architecture in this new approach. A more detailed presentation of this integration architecture can be found in Chapter 4.

Figure 3.3: Middleware composed of JADE and gateway federates

3.2.2 Communication Issues

This section mainly discusses about the communication issues of an agent-based distributed simulation. We present a conceptual classification of the communication involved in the proposed integration architecture which will be further investigated in the following chapters.

Figure 3.4 provides a conceptual classification of the communication that are required in the system. Commonly there are four kinds of communication involved in an agent-based distributed simulation:

1. **Between federates:** This includes sending/receiving interactions between federates, and updating/reflecting individual attribute values. (e.g. (1) in Figure 3.4). This kind of communication covers both the communication between two agent federates or the communication between an agent federate and a non-agent federate such as a separate environment federate.
2. **Between agent federate and agent:** This includes the interactions between an agent and its federate via its sensor and effector that are required for autonomous

Figure 3.4: Classification of communication

agents to perceive and react. (e.g. (2) in Figure 3.4).

3. **Between environment federate and GUI:** This provides a way for observers to interact with the virtual environment during the simulation execution, for example, by slowing down or speeding up the simulation, or by adding or deleting objects in the environment (e.g. (3) in Figure 3.4).
4. **Between agents:** This includes agent-to-agent communication. An agent may send information in specific formats that can be understood by other agent(s), such as ACL messages. (e.g. (4) in Figure 3.4). This kind of communication may occur between the agents in different federates, or even the agents in the same federate if the agent federate is constructed to support multiple agents.

In our proposed integration approach, the HLA/RTI plays the role of run-time infrastructure which enables the execution of agent federates and an environment federate to proceed concurrently exchanging data and synchronizing activities. However, an agent's

communication and computation is generally asynchronous, and it also has a distinct thread separate from its federate's, thus we need some effective mechanism to synchronize its activities with the agent federate. Chapter 4 will address the communication issue between the agent federate and agent. In Chapter 5, we will address the communication issue between different federates and provide the implementation of a prototype system with a GUI that also includes the third kind of communication.

Agent-to-agent communication is one of the key characteristics of MASs. When we integrate multiple agents into distributed simulations, communication is a very important issue that needs to be addressed. In simulations with multiple agents, it is very common for each of them to have only a limited view about the environment. The agents still need to communicate with each other to overcome the shortcoming of limited knowledge. Thus in Chapter 6, we will focus on this last form of communication, which occurs between the agents of simulations.

3.3 Summary

This chapter has introduced some basic techniques of parallel and distributed simulation. Distributed simulation enables participants located in possibly different geographical locations to share a common virtual world. Agent technology enables distributed simulation to achieve rapid and accurate decision making, and distributed simulation can increase the reusability and interoperability of agent systems. By combining these two technologies together, many benefits have been found for both sides. It is therefore attractive to integrate autonomous agents into distributed simulations.

The fundamental issues that need to be addressed for this project have been highlighted, and some problems of existing approaches have been presented. We have addressed the issue of how to integrate agents into an HLA-based distributed simulation.

We have also discussed about communication issues involved in an agent-based distributed simulation. These discussions are the foundations for a successful approach to the integration of agents into distributed simulations.

Chapter 4

Integration of Agents into Distributed Simulations

In Chapter 3, we have introduced our approach of integrating agents into distributed simulations. In an agent-based simulation, gateway federates are constructed as agent federates that take charge of agents. Using the HLA RTI, an agent federate can interact with a non-agent federate such as an environment federate. In this chapter, we present more details about our integration architecture.

4.1 Agent Architecture

Our target agent architecture is displayed in Figure 4.1. As we see, there are three main components in this architecture: sensor, deliberation and effector. Moreover, our agents are implemented as “situated” or “embedded” agents, that need to interact with an environment.

In order to enable the interaction between agents and their environment, we first need to develop two interfaces to provide interaction between an agent federate and its agent(s) (see Figure 3.4 (2) in Section 3.2.2), namely, a *sensor* object and an *effector* object. The sensor enables an agent to perceive the environment within which it is situ-

Figure 4.1: Our target agent architecture

ated, and the effector enables it to act upon the environment as a response. These two kinds of interaction interfaces are opposite in information-flow directions, the sensor for *environment* $\rightarrow$ *agent* and the effector for *agent* $\rightarrow$ *environment*. With these facilities, a complete interaction for a situated agent is obtained, and a three-phased (sense-deliberate-react) agent simulation cycle can be achieved.

According to our target agent architecture in Figure 4.1, we have the logical structure of an agent simulation cycle as shown in Figure 4.2. The execution of an agent will proceed in three logical phases: sense, deliberate, and react, and this cycle will be repeated during the simulation execution.

Figure 4.2: Logical structure of an agent simulation cycle

4.1.1 Sense

In the first “sensing” phase, each agent’s internal database is updated according to the information it gets from the sensor interface. In our simulation, each agent has an internal data set which can be regarded as its private *memory* or *knowledge base*. The data set holds the agent’s model of the environment obtained from the sensor. The internal data is private in the sense that other agents have no authority to access it directly. Compared with the agent’s internal data set, the data that is conceptually visible to other agents in the environment is called external data.

Obviously, what an agent will get to know about its environment depends on its sensor (in some cases, ACL messages from other agents are available alternatively). The sensor translates the environment’s facts into information that can be understood by the agent, and different sensors can have different functions for perception. This is very similar to the way we look, smell and touch via our eyes, nose and hands respectively. No doubt an agent with a more powerful sensor is endowed with stronger capability for getting information from its environment.

Generally, a sensor provides its agent with the status of its neighborhood including whether there are any objects nearby and what these objects are. We define every agent’s sensor to perceive all the objects in the close neighborhood, so the field of agent vision is limited to a small distance. It enables agents to touch and see within a limited bound. We could also enlarge the sensor range to let the agent get more visible facts/attributes of the environment, offering further and deeper vision for the agent.

4.1.2 Deliberate

The second “deliberating” phase involves the control algorithm for decision making and action selection. The new facts received will be matched against the event-option rules

which constitute the agent's state transition system. After the deliberation phase, a plan will be adopted and particular behavior will be triggered to determine what actions will be performed by the agent.

In our system, the behaviors of an agent are controlled by a Finite State Machine (FSM), a finite set of states and its transition map being predefined in the agents. Once a state-event pair is satisfied by the current condition, transition to another state is made and actions are performed. For example, when an agent takes the behavior of searching for an object in an environment, it will match its perception every time it steps into a new position. If no object is detected, it will go on roaming; if one is found, it will start to perform the next action such as moving this object.

The agent's deliberation defines its personality, such as the simple intelligence of avoiding obstacles and approaching something. The agent is autonomous in the sense that it is not under the control of a human being, and all its decisions are derived from the embedded rules and the information accessible to it. More particularly, the agent's internal states and the set of rules determine the agent's next possible influence upon the environment which is written in the effector.

4.1.3 React

After the internal deliberation determines the actions to be performed in the next phase, the corresponding effector will be activated. The final "reacting" phase involves performing the actions queued in the previous "deliberating" phase. This procedure utilizes the effector which is an interface to the environment as well as the sensor. The actions taken by the effector will affect some attributes of the environment, for example, the position of the agent, the existence of an object in the environment.

The sensor/effector objects transferred between the agent federate and the agents

endue the agents with the ability to interact with the environment, not only perceiving the environment but also influencing it. Thus with the sensor and the effector in place, interfaces between agents and its environment have been well defined. The sensor and effector differ in that the former is in charge of querying and retrieving information from the environment, while the latter is for agents to act upon the environment in response. These two are somewhat similar because they are both interaction channels between the agent and its environment. Their data-flow directions are opposite.

In addition, those actions such as sending messages to each other for achieving cooperation among agents can also be comprised in the last phase, because sending messages can also be considered as a way of influencing the environment. It is possible to endue the effector with the function of sending messages to other agents in the mode of either broadcast model or point-to-point model. This function will enable agents to collaborate with each other in groups for getting a better working efficiency.

4.2 Overall Architecture

As shown in Figure 4.3, we have proposed an overall architecture for a MAS in an HLA-based simulation. The environment in which agents are embedded is represented by a distinct federate. Gateway federates which are agent federates and the environment federate which is a non-agent federate are connected using the HLA RTI. In this architecture, every agent is attached to a gateway federate. We can have several agents in a group attached to the same gateway federate or a single agent attached to a separate gateway federate. At the same time, different federates can also be placed in the same machine or different machines.

Figure 4.3: An architecture for a MAS in an HLA-based simulation

4.2.1 JADE

We choose Java as our development language because it is inherently designed to support multiple host architectures and it has many attractive features, particularly geared towards object-oriented programming in distributed heterogeneous environments. As mentioned in Section 3.2.1, JADE [7] is a well-known MAS development toolkit supporting the FIPA specifications, an internationally agreed agent standard, thus we select it to support our agents and their communication. JADE is also a software framework fully implemented in the Java language.

This standard platform contains FIPA agent management components, and some agents that support general agent specific services for all the other agent entities. Based on the FIPA agent management reference model (refer to Figure 2.4), JADE consists of the following logical components, each representing a capability set, including agent platform (AP), agent management system (AMS), directory facilitator (DF), message transport system (MTS), and remote monitoring agent (RMA).

In JADE, the AP is the physical infrastructure in which agents can be deployed. The AMS is responsible for managing the agent platform and providing the white-page services. The AMS also maintains a directory of *agent identifiers* (AID). The DF provides the default yellow page service in the platform. The MTS of JADE is also called *agent communication channel* (ACC) sometimes. It is a software component controlling all the exchange of messages within the platform and to/from remote platforms. The RMA is a special kind of agent that provides a graphical console to monitor and control the JADE platform and, in particular, the life-cycle of agents.

JADE can simplify the development of multi-agent systems through a set of tools that support the debugging and deployment phase. Its AP can be distributed across machines even those with different operating system as shown in Figure 4.4. This shows the JADE platform distributed over several containers where each container is on a different machine. JADE can have multiple agent containers where agents reside. One of these agent containers is the main container containing the AMS and DF and where the RMI registry, that is used internally by JADE, is created.

JADE's communication architecture offers flexible and efficient messaging, where JADE creates and manages a queue of incoming ACL messages, private to each agent. Agents can access the queue via a combination of several modes: blocking, polling, time-out and pattern matching based. In JADE, the full FIPA communication model has been implemented with its components fully integrated: interaction protocols, envelope, ACL, content languages, encoding schemes, ontologies, and finally, transport protocols. The transport mechanism, in particular, is claimed to adapt to each situation, by transparently choosing the best available protocol. JADE supports user-defined content languages and ontologies for agent management.

Another valuable feature of JADE that needs to be mentioned here is that Java applications can use it as a library. An in-process interface has been implemented, allowing us

Figure 4.4: JADE agent platform distributed over several containers

to launch a JADE runtime from an external application. This mechanism is flexible, and it allows several different configurations for a JADE platform, such as a complete in-process platform composed of several containers on the same Java Virtual Machine (JVM), and a hybrid platform partly in-process (containers launched by an external Java application) and partly out-of-process (containers launched from the command line). This potentially offers the possibility to advance the simulation into hybrid platform mode for further investigation.

In addition, JADE has a very flexible strategy for the agent rule engine. We can use the finite state machine supported by JADE to create behavior scheduling for an agent. Alternatively, in some complicated cases, it is also acceptable to exploit reasoning capabilities, because JADE has also been integrated with JESS (Java Expert System Shell) [19], a Java shell of CLIPS.

4.2.2 Gateway Federate

Like other federates in a federation, including non-agent federates, each gateway federate that hosts agent(s) has one point of attachment to the RTI. The gateway federates access the RTI via the *RTI ambassador* and *Federate ambassador*. Both of these ambassadors are part of libRTI, the library that a federate uses to invoke the HLA services (see Figure 4.5). However the federate ambassador is abstract, thus the gateway federate needs to implement the functionality declared in it.

Figure 4.5: RTI ambassador and federate ambassador

By using gateway federates, it is possible to utilize the services of the HLA RTI to meet the requirements of an agent-based distributed simulation:

- **Creating and Deleting:** An agent-based distributed simulation can be created or destroyed as a federation.
- **Joining and Resigning:** A gateway federate can join a federation or resign from a federation together with its agent(s). During the federation execution, more than one gateway federate can join the same federation at any given time, and then execute synchronously with all the other federates.

- **Publication and Subscription:** Gateway federates can declare their interest in the information of the environment or other gateway federates, and publish information they will present to others. The information can be information from/to other agent federates or non-agent federates, including attributes and interactions sent among these federates.
- **Registration and Discovery:** Newly-joined agents can be registered by a gateway federate, and discovered by other federates including agent federates (gateway federates) or non-agent federates (environment federate) during the federation execution. Similarly the environment federate may register objects that are discovered by the gateway federates.
- **Attribute Update and Reflection:** When one gateway federate updates an attribute, data can be reflected to other gateway federates or the environment federate. On the other hand, when an attribute is updated in the environment, data can be reflected to gateway federates.
- **Sending and Receiving:** Messages between agents can be sent from a gateway federate and received by another one in the form of interactions transferred via the HLA RTI.
- **Time Advance Request:** A gateway federate can request to synchronize its activities with others during the federation execution, using the time management services discussed in Section 3.1.3.2.
- **Data Distribution Management:** Gateway federates can indicate their interest in receiving or providing updates, in order to reduce unnecessary information to be transferred among the gateway federates and the environment federate. The DDM services described in Section 3.1.3.3 help to support efficient routing of data and lighten the heavy burden of the network.

- **Ownership Transfer:** In a federation where multiple agents are executing concurrently, a gateway federate must get the ownership of an attribute in the environment as the permission for its agent(s) to alter the attribute. An attribute can only be owned by one gateway federate at a time, to ensure the consistency of the distributed simulation.

These services' relations between the gateway federate and the federation are shown in Figure 4.6.

Figure 4.6: Relations between a gateway federate and a federation

4.2.3 Simulation Cycle

In our system, the agents and the environment are represented by different federates connected by the RTI. These federates will execute synchronously with each other. None of

them can advance its simulation time until the other federates request a time advance too. Although the agent executes in a separate thread, it is still kept at the same simulation time as its gateway federate. The simulation is a synchronous, distributed, time-driven system where simulation advances in timesteps.

Figure 4.7 illustrates our initial design for the simulation cycle of a federation containing one gateway federate and the environment federate. Although only one agent is shown in the figure, we could also allow several agents to execute in the same gateway federate. When the simulation starts, the environment federate initializes the objects in it. After the initialization is completed, the environment will loop waiting for any information that comes from the gateway federate. Each gateway federate can initialize its agent(s) in a launched JADE agent container. After all the agents are configured, the federation will start execution according to the simulation cycle.

In each simulation cycle, first the decision of an agent is written to the effector, such as its new position and the intended modification of the environment (step 1). This will be read from the effector by the gateway federate. Then the gateway federate sends all this information to the environment federate via the HLA RTI (step 2). When the environment federate gets the information from the gateway federate (step 3), it will start to carry out some work modifying the environment according to the information, such as placing an agent into a new position, and adding/deleting objects in the environment according to the agent's decision (step 4). After that, the information about the newly modified environment is available for the agent side, and sent out via the HLA/RTI by the environment federate to the gateway federate (step 5). The gateway federate receives this information (step 6), and then sends it to the agent via the sensor (step 7). The agent uses the information provided by the sensor and its own knowledge base, then deliberates according to its rules and makes a new decision (step 8). Then a new effector will be generated, and the information is ready to be sent by the gateway federate to the

Figure 4.7: The simulation cycle of a federation

environment federate again.

4.3 Linking the Gateway Federate and Agent

In our simulation, the gateway federate and the agent are still different threads that proceed concurrently. Section 4.1 has stated that every agent has two inherent components tightly associated with its gateway federate, namely, sensor and effector, which endow the agent with the capability to perceive and act upon the environment in which it is embedded. These two interfaces differ from the agent's internal process of deliberation for their accessibility of external data or information of the environment.

4.3.1 Java RMI vs O2A Communication Channel

When investigating a feasible implementation for the proposed overall integration architecture, a problem arises about how to facilitate the agent interaction interfaces between the gateway federate and agents. Two alternative approaches are discussed in this section.

In the first approach, the interaction interfaces can be constructed with the assistance of Java Remote Method Invocation (RMI). RMI in Java provides a distributed object model that crosses JVMs seamlessly. It enables developers to treat remote objects and their methods very much like normal Java objects in distributed applications. In this model, the FIPA agent platform and the gateway federate are placed in distinct JVMs. The agent platform offers standard agent-specific services. Agents living in agent containers interact with their environment via RMI. The agent side is the client from the aspect of RMI. By contrast, the software in the gateway federate is the server. The software in the client and server includes interfaces such as sensor and effector. The framework of this approach is shown in Figure 4.8.

This approach is easily and widely applicable, but it lacks efficiency for utilizing two JVMs. Although an agent will execute as a separate thread from the gateway federate, it should be more appropriate if we could have it running in the same JVM with its gateway federate. Because of the latency of transferring messages between JVMs, using one JVM can bring better efficiency than using two JVMs. This leads us to find a new approach of connecting agents and the gateway federate. We require a mechanism to transfer both of the two interactions between the gateway federate and the agent, in a single JVM instead of using Java RMI. This can be achieved by using the Object-to-Agent (O2A) communication channel that is available in JADE.

As shown in Figure 4.9, in this second approach, the gateway federate is placed in the same JVM with the agent. Since JADE allows an external application to start a runtime,

Figure 4.8: Two-JVM approach

Figure 4.9: One-JVM approach

we start an agent runtime and control agents directly from the federate's JVM utilizing the JADE library. Thus in the implementation, we launch an agent by an external Java program instead of by the JADE platform itself. We obtain a good framework in which the agent executes in a separate thread while having the same JVM as the gateway federate.

The agent must have declared its willingness to accept passed objects from other software components running within its JVM. After that, the gateway federate passes application-specific objects to a local agent via the O2A communication channel. Since the O2A communication channel can only deliver objects from the gateway federate to the agent, the gateway federate will continuously send sensor and effector objects to its agent alternately according to the agent simulation cycle described in Section 4.1. Such transmissions could be done asynchronously, with a non-blocking call, returning immediately after putting the object in the agent's internal queue, or synchronously, with a blocking call that does not return until the agent retrieves the object from the private queue. In our prototype, we have demonstrated that the agent can interact with its environment via such communication channels successfully.

We have compared the latencies of transferring messages for these two approaches on a single computer configured with Pentium 4 CPU and 512MB of RAM. A *String* object with a value indicating the time of sending the message is transmitted from one side to the other. The receiver side prints out the transmission time of sending this message by deducting this time from the time of receiving the message. Then we can find out how long it will take for the transmission for these two approaches. Figure 4.10 (a) shows the latencies in the two-JVM approach using JAVA RMI, and Figure 4.10 (b) shows the latencies in the one-JVM approach using the O2A communication channel. We test messages with increasing lengths from 10 to 10000 bytes. Clearly the second approach always spends much less time in transmission than the first approach.

Figure 4.10: Latencies of sending messages: (a) via JAVA RMI; (b) via JADE O2A communication channel.

In conclusion, the one-JVM approach gains much higher efficiency in performance than the two-JVM one. This is the reason why we will select it for our proposed framework. However, the two-JVM approach could be used in cases where an agent platform does not support an external runtime, or an O2A communication channel is not available.

4.3.2 Synchronization using Latch

As discussed previously, Figure 4.2 shows the agent simulation cycle, and Figure 4.7 shows the gateway federate's simulation cycle. Now we will discuss about how to combine these two simulation cycles together and make them match, using both the approaches described in the previous section.

It is not difficult to provide a connection using the RMI approach, as shown in Figure 4.11 (a). The gateway federate gets information from the environment federate after a time advance request is granted, and then equips a sensor for the agent. After the agent's deliberation, a decision is made about the next action, and an effector object carrying this information will be passed to the gateway federate. The gateway federate

receives the updated information from the effector, and then requests a time advance to transfer these updates to other federates such as the environment federate.

In the first approach, the RMI itself is a synchronous way of transferring information. However in the second approach, we need to consider more about synchronization. Because the O2A communication channel is a one-way channel for objects, the gateway federate needs to continually send sensor and effector alternately to an agent, carrying perception and reaction information between the agents and environment. Every time a gateway federate puts an effector to an agent, it should wait until the agent completes the current action which modifies the effector. Since an agent's deliberation is usually a time and space consuming process whose execution time cannot be predicted [58], we must ensure that the agent can always take actions in time after making a decision. It is not acceptable if an agent acts like a man living in the past. For example, an agent might move to a new position in the environment, but it still tries to use a previous effector to alter surrounding objects. Neither is it acceptable if an agent's actions are not in sequence when received by the environment. Obviously, for reasons of consistency, it is necessary to investigate some synchronization strategy for our integration proposal.

To solve these problems, we need to introduce a condition variable called a *latch* object into the system. Figure 4.11 (b) shows the synchronized simulation cycles of a gateway federate and an agent using a shared latch object as the synchronization point. When a new agent is initialized, a prepared latch object will be passed to it as an argument at the same time. We can let the gateway federate wait on the latch until the agent signals the latch. With the employment of the latch, the federate thread is conveniently blocked and released at appropriate points. In this way, we utilize the effector events as synchronization points, and at other times, all threads proceed concurrently. As the latch object is shared by the gateway federate and the agent in a single JVM, this synchronization mechanism involves minimal overhead.

Figure 4.11: Linking the agent and gateway federate: (a) the two-JVM approach; (b) use a shared latch object as the synchronization point in the one-JVM approach.

We have discussed the situation in which just one agent is involved. When multiple agents are activated in the same federate, it will be necessary to use multiple latches as shown in Figure 4.11 (b). The use of a latch with multiple agents is the same as the situation of a single agent. However, in order to speed up the simulation, it is better to enable agents in the same gateway federate to execute concurrently rather than executing in sequence. In each simulation cycle, instead of sending a sensor and an effector alternately or waiting on latches one by one, the gateway federate can send all the sensors, then all the effectors at the same time. Before the gateway federate requests a time advance, it must have received all the signals from these agents. In this way, although there might be multiple agents activated in one gateway federate in the simulation, the federate can still execute in the same way as if there is just one agent activated.

4.4 Summary

In this chapter, we have constructed a general overall architecture for integrating agents into distributed simulations. A special kind of federate, called a gateway federate has been introduced to provide seamless connections between a FIPA-standard agent platform and the HLA RTI. The simulation will be advanced according to a simulation cycle. Two approaches to supporting agent interaction with their environment have been presented and compared, using Java RMI or O2A communication channel. Synchronization between the gateway federate and agents has also been investigated here. We have used a latch as the synchronization point in the second approach to solve this problem.

Chapter 5

Mine Sweeping Game

A prototype system is required for testing the feasibility of the architecture described in the previous chapter for integrating agents into distributed simulations. For this reason, we implemented a prototype system called Mine Sweeping Game (MSG). The main objective of the prototype is to enable multiple agents to perform tasks in a shared virtual environment. This prototype is constructed as a test-bed, and it is used as a basis for further investigations in agent communication issues and as a way of evaluating performance. During the process of the implementation, a more direct understanding of research issues involved in the research area is obtained and valuable experiences are gained.

5.1 Description of the Game

The MSG draws some valuable attributes from some previous simulations such as stone-picking robot [12, 14], TileWorld [45], MiMaze [22] and Maze [11], a game derived from MiMaze. MiMaze is a distributed multiplayer game on the Internet developed by Gautier and Diot. Maze developed by Cai et al is constructed upon the HLA, and both the stone-picking robot developed by Hirsbrunner et al and the TileWorld benchmark used by Lees et al used agent toolkits. However, none of these games addresses the problems of integrating a multi-agent framework into a HLA-based DVE in the same way as we do

in the MSG prototype, using the gateway federate and FIPA-compliant agent toolkit.

In Maze, the environment in which multiple avatars are embedded is distributed and represented by four federates. These federates interchange information and messages through the HLA architecture. The stone-picking robot scenario gives a new way to handle emergent cooperation situations in a multi-agent system. The TileWorld simulation as implemented by Lees et al [35] gives us a good example of implementing an agent together with the HLA. It has two federates displaying the environment and the agent respectively, which we also adopt initially. Their main tool is SIM-AGENT, which is a sequential, centralized, time-driven simulation component for multi-agent systems [38].

Our MSG prototype is developed as a proof of concept of the generic framework for distributed agent based simulation proposed in this thesis. Different simulation issues should be addressed when integrating multiple FIPA-compliant agents into distributed simulations. Thus different features of the MSG are designed to meet different requirements of our simulation. The MSG can be used to investigate various research issues including multiple agents embedded in an environment, agent communication, agent cooperation and coordination, hierarchical agents and their aggregation and disaggregation. None of the available benchmarks provide all these features. These are the reasons why we develop our own prototype system.

In the MSG, a battlefield is built in which multiple soldiers take as their goals to clear all the mines in a particular area so to keep it safe from danger. Here each soldier is represented by an agent, which is human-like and can act autonomously. The soldiers interact with the environment, get information from their sensors, make deliberations and then act upon the environment as responses through their effectors. They must be able to avoid obstacles such as trees and rivers in the battlefield, and even deal with the situation where the battlefield is fast changing, and can be modified by human participants during the simulation execution. The obstacles can be changed according to time or be influenced

by observers/users of this simulation. For example, a mine could blast after a specified period of time, or obstacles could be moved by a human participant who is observing the game via the GUI. In the environment, soldiers, mines, trees, rivers and borders are all implemented as objects, among which trees, rivers borders and mines are called obstacles. Mines are also treated as obstacles, because soldiers cannot step on them either. The game is over when there are no mines left in the area, in other words, when the soldiers have completed their mission successfully.

The whole system is structured upon the HLA/RTI, and it consists of several distributed federates including more than one gateway federate and an environment federate. The HLA/RTI plays the role of simulation engine (SE) which advances simulation time in ticks. The framework of the prototype system is shown in Figure 5.1.

Figure 5.1: Framework of the MSG

At the start of the simulation, there are dozens of mines distributed in the battlefield which is comprised of rivers and trees as static obstacles together with the moving soldiers. Borders are defined along the edge of the area where soldier agents will clear the mines.

Soldiers roam in the battlefield, finding out if there is any mine left in the area. If a soldier discovers a mine, he picks the mine up, adds it into his pack and then approaches the borders to throw it away out of the borders. Each soldier can be assigned to be able to carry one or more mines at a time, so he will not pick up more if his pack is full. Instead, what he could do is to call another soldier to come and deal with the detected mine. In this case, the soldiers are not just working together toward the same aim, but also coordinate with each other. Soldiers can be grouped into teams and configured to broadcast their discoveries to their teammates. It is also possible to introduce different types of agent groups with different goals and integrate them into the same environment.

In our prototype, none of the soldiers has entire knowledge about the battlefield. Each of them has only a limited view around him. The agents must have good capability to deal with various situations without human assistance. They should have the ability to face a flexible condition and settle problems such as which direction to choose to avoid obstacles and what to do next according to the partial information from the battlefield. At the same time, they should be enabled to communicate with each other, as well as interacting with the environment. Agent communication will be further discussed in Chapter 6.

5.2 Soldier's Behaviors

We use JADE to implement the behaviors of soldiers in the MSG. Commonly, an agent is able to carry out several concurrent tasks in response to different events. In order to make agent management efficient, every agent is composed of a single execution thread and all its actions taken towards a task are modelled as concurrent behaviors. We have defined and initialized several behaviors and sub-behaviors for our MSG soldier agents. Soldiers are implemented as a set of behaviors, and these behaviors are scheduled using FSM-based scheduling as introduced in Section 4.1.2. During execution, the scheduler will

Figure 5.2: Soldier behaviors in the MSG

carry out a round-robin non-preemptive scheduling policy among all behaviors available.

Figure 5.2 displays the set of behaviors and their relationships for the soldiers in the MSG scenario. First of all, a soldier agent without a mine in his pack executes the behavior *roam and detect mines*. As soon as he discovers a mine around, he stops and executes the *pick up mine* behavior as the next behavior. If the soldier succeeds in the *pick up mine* behavior, i.e., if he has put a mine in his pack, he will execute the next behavior of *reach border*. Alternatively, if he fails in *pick up mine* behavior, the subsequent behavior will be the first behavior instead. That means, in this case, he should go back to continue to detect another mine. For example, if two soldiers try to pick up the same mine at the same time, only one of them can succeed. Usually, the soldier agent can pick up mines successfully without conflict arising from other soldiers and constraints from the environment. Then after reaching the border, the soldier throws the mine away out of the particular area, namely, deletes it from the area by taking the *discard mine* behavior. Then, the soldier goes back to the original behavior again, wandering around searching for mines.

Some behaviors are composed of more than one sub-behaviors because they have more operations to be performed and more sub-tasks to be completed before a high level task is completed. For example, considering the MSG scenario, a game without too complicated agents, if a soldier has no mine in his pack, he will continuously wander in the battlefield. This behavior *roam and detect mines* includes the operations of perception

about the information of the environment around the soldier, selecting a new direction to go towards when encountering obstacles, and even considering options about the mines if several are detected at the same time.

Using the JADE toolkits, we have designed and implemented the behaviors and sub-behaviors of MSG soldiers. The following description will go into particulars one by one.

5.2.1 Roam and Detect Mines

Figure 5.3 indicates how the *roam and detect mines* behavior is implemented. It is comprised of six sub-behaviors. Firstly, the agent gets perception from its sensor which is called *read sensor*. The second sub-behavior is to consider whether there are any objects around. The objects here involve all things including trees, rivers, borders, mines, and even other agents. If there is not any object in the neighborhood, the agent directly goes to the next sub-behavior to wander around and then reads the sensor again. If there is an object, the agent will identify if it is a target object — a mine, which is the fourth sub-behavior based on the agent's perception from its sensor information. If not, the agent will select a direction to go avoiding obstacles, which is the fifth sub-behavior; otherwise, it is in the last state of *ready to pick up mine* which is the sixth sub-behavior, and will progress to another behavior — *pick up mine* behavior.

In the MSG, each soldier has only a limited view of the battlefield around him. For example in Figure 5.4, the soldier on the left side perceives nine grid cells around him including the center one where he stands, and finds three trees in his neighborhood. The soldier on the right side can only perceive nine grid cells around him too, but he detects a river. This kind of sensor information can be simply represented by character arrays, such as $\{nil, t, t, nil, s, nil, t, nil, nil\}$ and $\{r, nil, t, r, s, nil, r, r, nil\}$.

Figure 5.3: Roam and detect mines behavior

Figure 5.4: Sensor range of soldiers

A soldier has nine options for each next step including standing still if and only if no obstacles are nearby (Figure 5.5). The situation with river and trees around is more complex because of avoiding obstacles. A soldier should not step on objects including mines and other soldiers, otherwise some objects may have the same position which is not allowed in the environment. He will randomly select the next available position. Because of the flexibility of the environment and the soldier's capability of avoiding obstacles, we cannot predict his route in advance.

Figure 5.5: Agent direction choice

5.2.2 Pick Up Mine

Figure 5.6 shows an example of the trigger for this behavior. When a soldier moves to a new position and he discovers a mine in his sensor range, he will try to pick the mine up if his pack is free. As shown in Figure 5.7, this behavior is much simpler than the *roam and detect mines* behavior. Generally, we allow a soldier to carry just one mine at a time, so in this case, he may need to send information to another soldier who is free and can come to clear this mine instead of him. This involves the issue of agent communication which will be discussed in the next chapter. The agent should distinguish the case where there is only one mine from that where there are several mines. If the soldier discovers several mines at the same time and he has the constraint that he can take just one mine in his pack, the soldier will select randomly from these several mines.

Figure 5.6: Soldier discovers a mine when moving to a new position

It is possible that a soldier may fail in picking up a mine if another soldier attempts to pick up the same mine at the same time (in the same simulation cycle). So in the next simulation cycle that starts with reading a sensor, the soldier will get to know whether he succeeds in the previous action. If he succeeds in picking up the mine, he will add it into his pack and start the next behavior of *reach border*. Otherwise, he will execute the behavior of *roam and detect mines* again.

Figure 5.7: Pick up mine behavior

5.2.3 Reach Border

Figure 5.8 illustrates how the *reach border* behavior is implemented. The soldier carrying mine/mines will move towards the nearest border of the area by calculating according to his current position and the knowledge about the environment. Six sub-behaviors fit together for this behavior. The *read sensor* sub-behavior is first executed. If the soldier detects an object nearby, he will compare and decide the next step which will make him nearer to the border. Otherwise, without an object around, he just needs to go towards the border directly. If there exist several positions that can make him nearer to the border, he will randomly choose one of them. If the soldier cannot find a position to make himself nearer to the border for the next step, he should also randomly choose a position to go subsequently. If the soldier reaches the border, he progresses to the *discard mine* behavior. Otherwise, he continues to approach the border holding mine/mines in his pack. During this phase, the soldier is also able to discover new mines, but if his pack is already full, he will not pick up any more.

5.2.4 Discard Mine

As shown in Figure 5.9, *discard mine* is probably the simplest behavior of soldiers. The soldier that brings a mine to the edge of the area will clear the mine from the area. In this case, the mine disappears, and the agent becomes free and is ready to clear another mine. The *roam and detect mines* behavior will be reactivated again.

5.3 Implementation of the MSG

In the MSG prototype, all the federates in the federation including the gateway federates and the environment federate are implemented as time-regulating and time-constrained. Currently we are using a time-stepped mechanism for the time synchronization. All of

Figure 5.8: Reach border behavior

Figure 5.9: Discard mine behavior

the federates must have their time advance requests granted through the RTI before they can advance their local time.

5.3.1 Federation Execution

There exist several ways to implement the MSG prototype system and to execute the simulation cycle illustrated in Section 4.2.3. One simple way is to send interactions among federates in the MSG federation to enable them to exchange data. In this way, sensor or effector data can be received by the other side when a time advance request (TAR) is granted. Because an agent's new position needs to be reflected to the environment federate before the environment sends interactions about the information around the agent back to the agent federate, the original approach needs two time-steps for each simulation cycle to be executed, as shown in Figure 5.10.

Figure 5.10: Federation execution using sending interactions

We have developed an alternative way to implement the prototype, using the data

distribution management (DDM) services provided by the HLA/RTI, which can reduce unnecessary information to be transferred between the federates by filtering out irrelevant data [61]. In this way, DDM services are used to provide relevant information to construct the sensor, and interactions are no longer used. Obstacles and mines are treated as objects in the federation, and ownership management (OM) services are utilized for management of these objects. For example, once a mine is picked up by a soldier, the object ownership of the mine will be transferred from the environment to the gateway federate of the soldier agent, then the soldier agent is responsible for updating the position of this mine, or even deleting it from the federation. As the RTI ensures that each object has a unique owner, this technique can be used to resolve conflicts such as two soldiers attempting to pick up the same mine at the same time.

In the alternative approach, each federate first needs to initialize its subscription and update regions. Soldiers are moving, thus they have dynamic subscription region for the limited view of the environment. Firstly, the agent federate gets the initial position of the soldier from the effector, sets the subscription/update regions that includes nine grid cells centered on it, and updates the agent position to the environment federate. In each simulation cycle, if there are discovered objects for a soldier, which means the objects are within the subscription region of his agent federate, the agent federate will process this information and construct a new sensor for the soldier. When the agent gets the sensor, it deliberates and writes the effector. After that, the agent federate will immediately modify the subscription/update regions according to the agent's new position using the DDM services described in Section 3.1.3.3 and then updates the position to the environment federate. More details about how an agent federate modifies the subscription/update regions can be found in [61].

As shown in Figure 5.11, this approach still needs two time-steps for a simulation cycle to alter the agent's position and the corresponding subscription region. One problem with

this approach is that when the next time-step begins, the agent federate may receive outdated reflected objects because of the dynamic subscription regions. If in Figure 5.11 (a) occurs before (b), namely, the environment federate updates attribute values before the agent federate has modified the subscription/update region according to the agent's new position, the agent federate may receive objects within the old subscription/update region, which are outdated and should be discarded.

Figure 5.11: Federation execution using DDM services

To solve this dynamic region problem, we enlarge the soldier's subscription region. For example, in Figure 5.12, a soldier's subscription region is configured to be as large as twenty-five cells centered on the soldier's position instead of nine. No matter which direction the agent will go for the next step, the enlarged old subscription region can still provide enough information for the soldier's vision that is limited to the nine grid cells in the neighborhood. Figure 5.13 shows the federation execution using DDM with an enlarged subscription region. In this way, it is possible to use just one time-step for

each simulation cycle instead of two, which speeds up the execution of the simulation. Figure 5.14 shows the execution times for 5000 simulation cycles for the three versions¹.

Figure 5.12: Enlarged subscription region of the soldier

5.3.2 Agent Behavior Scheduling

According to the descriptions in Section 5.2, we have combined the soldiers behaviors as shown in Figure 5.15. We can see that in this scheduling, operations upon the sensor and effector occur alternately, no matter how many loops are expanded before the agent progresses to another behavior/sub-behavior. Every simulation cycle starts with a sensor and ends with an effector. This ensures the scheduling matches with the simulation cycle of the federation as discussed in Chapter 4.

For enabling agent communication, a message receiving behavior can be implemented as a parallel behavior to all the other behaviors of a soldier. The additional behavior is always looping, because agents should be allowed to receive messages at any time. This will be explained in the next chapter.

¹More details can be found in paper [61].

Figure 5.13: Federation execution using DDM services with enlarged subscription region

Figure 5.14: Execution time of different versions

Figure 5.15: Flowchart of soldier behaviors

5.3.3 GUI of the MSG

Figure 5.16 shows the GUI of the resulting implementation of the MSG prototype system. This GUI is the main way of monitoring and altering the situation of the environment. We can initialize an environment with any size we wish. Using the buttons on the toolbar of the GUI, we can add obstacles such as trees and rivers in the environment. Mines and borders can also be changed. This facilitates the setting of variables and testing for the simulation. The busy soldiers who are holding mines and approaching borders are in red, and the agents with free hands are in yellow. We can even speed-up/slow-down the simulation via a speed control by altering the speed of the time advance of the simulation. In this way, the prototype system is a scaled real-time simulation as described in Section 3.1.1. The tick counter gives us a view of the elapsed simulation cycles, and the end time can be specified before the simulation starts.

5.4 Summary

We have implemented a prototype system to demonstrate the feasibility of the proposed overall architecture proposed in Chapter 4. The prototype system is a simple game in which several soldier agents try to complete the tasks of clearing all the mines out of an area. In the implementation of the prototype system, all federates are implemented as time-regulating and time-constrained, including both agent federates and the environment federate. The federation uses ownership transfer for the management of dynamic mines. DDM services have been used to provide filtered information to construct the sensor and control the unnecessary data placed on the network. However, using a simple approach, each simulation cycle still needs two time-steps, which reduces the efficiency of the federation. An enlarged subscription region has been applied to solve this problem.

Figure 5.16: GUI of the MSG prototype

Chapter 6

Agent Communication in Distributed Simulations

MASs have not only the advantages of distributed and concurrent problem solving, but also the advantages of sophisticated patterns of interactions including [31]:

- cooperation — working together towards a common aim,
- coordination — organizing problem solving activity so that harmful interactions are avoided or beneficial interactions are exploited, and
- negotiation — coming to an agreement which is acceptable to all the parties involved.

These patterns of agent interactions are achieved by communication among agents. Through communication, agents could be more comprehensible to each other and better achieve their design objectives.

Since communication is such an important issue of agent research, it is regarded as one of the key characteristics of MASs. In our investigation of agent-based simulations, we also need to address the issue of agent communication, and to solve some problems involved, especially when agents are integrated into distributed simulations. In this chapter, we

will present our work of achieving coordination through communication among agents.

6.1 Causality and Message Ordering

A basic requirement of a distributed simulation is to ensure that the individual simulation components perceive a common view of the virtual environment. Many papers have addressed this issue in recent years. A fundamental problem in large-scale distributed virtual environments (DVEs) is to maintain a consistent view of the simulated world among different simulation components [66]. For example, it is important that any pair of participants should perceive a set of messages/events in the same sequence.

Considering a collection of federates connected via a network, it is not difficult to give a scenario leading to a problem of consistency violation without an effective time management mechanism for ACL messages. Figure 6.1 illustrates a scenario about this problem. For example, suppose a scenario in which a treasure detector agent a discovers a gold bar, and sends a message to other agents including a picker named b and an observer named o , causing b to pick up this gold bar as a consequence. In the distributed simulation, the messages about the gold bar are as follows: the federate of a generates a message indicating where the gold bar is. This message is sent to all the other participant federates including b and o . Upon receiving this message, the gold bar will then be picked up in the federate of the picker b . Another agent o receives both the messages about the finding from the detector a and the picking up from the picker b . Delayed in the network, the message about the finding may possibly arrive later than the message about the picking up action. This causes the observer o to perceive the picking up action first before he knows some gold bar has been discovered. Is the gold bar still there? This gives rise to an inconsistency. It must be very strange for o that he gets to know a gold bar is there when it has actually been taken away. Thus to keep the causality of the system, messages should arrive in the correct order, that is, the message of discovering a

Figure 6.1: A scenario of distributed simulation leading to a violation of causality.

gold bar must arrive before the message of picking up the gold bar. This kind of problem is very common in distributed simulations.

Causality can always be kept naturally in the physical world, while in distributed simulations, it might be affected by many factors such as the latency a message encounters as it is transmitted through the network. Accordingly, a time management mechanism is required to control a distributed simulation to avoid violations of causality, especially for transmitting ACL messages between the agents embedded in the simulation. This can also ensure that repeated executions of the simulation with the same initial state and external inputs produce entirely the same result. So in our system in which multiple agents are embedded into a distributed simulation, we should not only accomplish the synchronization between different federates in the federation, but also a correct message ordering that will not violate the causality of agent communication.

6.2 How Agents Communicate using HLA

Considering the problem illustrated in Section 6.1, we should think about the communication of agents carefully when integrating them into distributed simulations. For the

requirements of distributed simulations, we must ensure the messages transferred among agents are perceived in the same sequence, namely, we must make a correct message ordering for them.

6.2.1 Agent Communication Route

For delivering ACL messages among agents, one possible way is to send them directly to those specified agents listed in the receiver slot of the ACL message, using the facility of the agent toolkit (see Figure 6.2 (a)). This can be achieved easily no matter whether those agents are in the same physical machine or not. But one of the HLA rules states, “during a federation execution, all exchange of Federation Object Model (FOM) data among federates shall occur via the RTI (rule 3)” [5]. According to this rule and the discussions in Section 6.1, obviously this way may cause some consistency violation and is inappropriate.

Thus to satisfy the requirements of the HLA and to guarantee the consistency of an application when it is distributed, we must choose an alternative way to send ACL messages. In this case, the communication between agents in different federates is made indirect via the RTI utilizing some mechanism instead of directly via the communication channel provided by agent platforms (see Figure 6.2 (b)). Each ACL message can be represented by an interaction sent between federates, and every gateway federate is given a mailbox as an accessory to take charge of ACL message transmissions. We develop the mailbox using the JADE agent toolkit and equip it with all the agent specific services. In this sense, we refer to it as a *mailbox agent* although it is simple and has only one behavior.

Figure 6.2: Agent communication route: (a) agents communicate directly via the communication channel provided by the agent platform (b) agents communicate indirectly via the RTI

6.2.2 Requirements and Constraints

In a time-stepped distributed simulation where multiple agents are activated, an agent may receive more than one message from different agents for a time-step. We should think carefully about these messages. Messages with the same time-step are considered to be sent at the same time, and we must make them received at the same time too. Because agent messages with the same time-stamp t are sent as interactions among federates at the same time-step t , they will be received after all federates advance their time from t to $t + \Delta t$. Thus to ensure a correct message ordering for a time-stepped agent-based simulation, in addition to the HLA rules such as rule 3 listed in Section 6.2.1, it is also required to make some additional conditions from the agent's point of view, such as:

- *An agent should have received all the messages of the last time-step before it can*

process them and advance to another time-step.

- *All the messages that are sent at the same time-step t should also arrive at the same time-step $t + \Delta t$.*

These requirements ensure that all messages will be received at the correct time-steps, and they will have a common message ordering.

According to the discussions in Section 6.2.1, we select the second way to send ACL messages indirectly via the RTI. To meet the requirements illustrated above, two constraints have been made on mailbox agents for agent-to-agent communication, they are:

1. *All outgoing ACL messages of an agent are passed to the local mailbox agent to be processed, and*
2. *A mailbox agent can only send ACL messages directly to local agents that are located in the same federate.*

In this case, those agents in different federates will not interact with each other directly and they use *mailbox agents* as communication middlemen. The first constraint ensures all messages are processed by the mailbox agent before being sent out. The second constraint ensures that a mailbox agent will not send agent messages across federates but only to agents in the same federate. Note that agents within the same federate have the same simulation time. With these constraints, we ensure that all exchange of FOM data among federates occurs via the RTI.

6.3 The Mailbox Agent

By using mailbox agents, four paradigms of communication can be supported among agents, classified from the perspective of the recipient(s) of a message, including:

- Peer-to-peer communication: an agent may call another one for help.
- Broadcast communication: an agent may broadcast some information about the environment to all the others.
- Group communication: an agent may send messages about its state such as “I am free” or “I am busy” to the agents that coordinate to work in the same group.
- Chain communication: an agent on calling for help can acknowledge the original sender that it will accept the task or not, or even transfer this task to another agent.

Note that all agents use *agent identifiers* (AIDs) to distinguish them in ACL messages.

A mailbox agent has two interfaces connected with its federate. One is *mailboxIN* that contains the received interactions from remote gateway federates, and the other one is *mailboxOUT* that contains the interactions to be sent to remote gateway federates. We can see that this is very similar to the situation of common embedded agents that have sensor and effector interfaces with their gateway federates.

All the incoming messages collected by a mailbox agent from local agents can be divided into three categories to be processed (Table 6.1). Here the term *local* indicates that the receivers are in the same gateway federate and the term *remote* indicates the opposite. It is also possible that an ACL message is sent to both local and remote agents.

Receivers	Category	Sent to agent	Sent as interactions
local agents	<i>isLocal</i>	yes	no
remote agents	<i>isRemote</i>	no	yes
local and remote agents	<i>isLocal & isRemote</i>	yes	yes

Table 6.1: Behavior of mailbox agent

6.3.1 Message Transmission Procedure

When an agent needs to send ACL messages to other agents, a complete transmission procedure consists of the following steps:

1. The agent passes the messages to the local mailbox agent (MB).
2. The MB collects all the messages from local agents.
 - (a) If the messages are sent to some local agents, the MB sends them accordingly.
 - (b) If the messages are sent to some remote agents,
 - i. the MB packs the ACL messages into the *mailboxOUT*, and passes it to the gateway federate.
 - ii. The gateway federate reads the *mailboxOUT*, and encodes each ACL message into a byte sequence which is then sent to all the other federates as a time-stamped interaction.
 - iii. Another gateway federate receives these interactions, decodes them into ACL messages and packs these messages again into the *mailboxIN* preparing for the subsequent transmission to its MB.
 - iv. The MB receives the *mailboxIN*, and unpacks it into the original ACL messages. The MB then delivers these messages to the local agents that the receiver lists may include.

3. Finally all the destination agents receive the ACL messages.

In this approach, we have ensured that a federation does not exchange data representing state changes of shared object instances or interactions outside of the RTI service, thus the consistency of the distributed application is not violated.

6.3.2 Synchronization between Federates and Agents

In our prototype, the RTI plays the role of the communication system for all the federates, providing the possibility for them to exchange data, and also to synchronize their activities. But the gateway federate and the agent entities including mailbox agents and common autonomous agents still have different threads that proceed concurrently. Thus a problem arises of how to synchronize these three threads of the gateway federate, the mailbox agent and the autonomous agents.

For this purpose, the condition variable latch should be utilized again, both between the mailbox agent and the gateway federate, as well as between an autonomous agent and the gateway federate. As we know, every latch is an object shared by an agent entity and its gateway federate. We let the gateway federate wait on the latch until the agent thread signals it, so that some synchronization points for consistency are established. Figure 6.3 shows how a gateway federate synchronizes with its mailbox agent and autonomous agents. The arrows show time dependency, for example, $m3 \rightarrow f2$ means $t_{m3} < t_{f2}$, i.e., there exists a time dependency between them. To ensure that all the sent ACL messages have reached the agents before the agents start to process them, we make an additional restriction, that is, to let the mailbox agent wait until it gets a return receipt for each sent message.

To enable the mailbox agent to process all the sent ACL messages of its agents, every agent is configured with a *myOUT* object when it is created. All the outgoing ACL messages of an agent will be put into its *myOUT* object. After all the agents of a gateway federate have done this and signalled on their *Latch*, the gateway federate will combine the ACL messages of these *myOUT* objects into a *mailboxOUT* object and send it to the mailbox agent. Then the mailbox agent starts to process these ACL messages according to Table 6.1. Some of these ACL messages are sent as interactions

Figure 6.3: Synchronization points

among federates and some of them will be added into a *sendList* object to be sent to local agents in the next simulation cycle. At the beginning of the next simulation cycle, the mailbox agent will get a *mailboxIN* object which carries the interactions received from other gateway federates, and then it adds the ACL messages whose receiver lists include agents of this gateway federate into the *sendList*. After that, the mailbox agent can send ACL messages to local agents according to the *sendList* and then clear it to be reusable in the next simulation cycle. Thus no matter whether an ACL message is sent to a local agent or a remote agent, the mailbox agent for the receiver will send it to

the agent at the next time-step. This ensures that the ACL messages sent at the same time-step t can arrive at the same time-step $t + \Delta t$ for both local agents and remote agents.

6.4 Applying ACLs in the MSG

As described in Chapter 5, currently a prototype system named MSG has been implemented, in which several soldier agents roam to detect and clear all mines out of an area so as to keep it safe. All the soldiers and the environment are represented by federates distributed in a local area network. They are connected by the RTI, and execute synchronously with each other, controlled by the HLA time management services. The prototype's main objective has been achieved by enabling those distributed soldiers to move in a shared environment performing tasks, and to communicate with each other freely.

6.4.1 Communication among Soldiers

We can adopt the interaction patterns mentioned at the beginning of the chapter in the MSG prototype. For cooperation, soldier agents can perform tasks individually and concurrently, each one detecting mines and trying to clear them out. Although they do not exchange information with each other during this procedure, they have the common aim to sweep as many mines as possible out of the area to reach the final goal that no mine is left. This case consists of agents simply working together and no communication exists among them. However, more valuable issues are involved in the latter two interaction patterns, coordination and negotiation. Since communication is a key capability which distinguishes MASs from other forms of software and provides the underlying power of the paradigm, our soldier agents have been enabled to communicate using some identified templates as illustrated below.

In the MSG, when a soldier moves and picks up a mine, he can inform the others about his status, such as where he is and how many mines he has cleared. In the situation that his pack is full and he cannot pick up any more, he may call other soldiers for help. The information about the new detected mine will be passed to a soldier who is free and who is the nearest to the mine's location¹. Then the free soldier will approach the mine and deal with it instead of the busy soldier. In this way, tasks of clearing mines may be transferred among soldiers, and a certain level of coordination is achieved among these agents. To enable this kind of communication, we have added a behavior for every soldier to receive messages as shown in Figure 6.4. This figure is the same as Figure 5.15, but with an additional behavior 5 that is parallel to all the other behaviors of a soldier. It is looping to allow agents to receive ACL messages at any time.

6.4.2 Message Templates

Figure 6.5 shows the behaviors of MSG soldier agents. After picking up a mine, if the soldier's hands become full, he will start to approach the border to discard the mines (out1). If he is still available to collect more, he can continue to detect and pick up other mines (out2). We can see that each soldier has two states:

- *available* — the soldier is able to pick up mines, such as in the behavior of roaming and detecting mines.
- *busy* — the soldier cannot pick up any more mines, such as when his hands are already full and he is approaching the border.

In the available state, a soldier is able to accept the tasks assigned by other busy soldiers who discover new mines.

¹A soldier knows which free soldier is nearest to the mine because a freeHands hash table is kept by each soldier. The details are provided in Section 6.4.3.

Figure 6.4: Flowchart of soldier behaviors with communication

Currently in our MSG policy, we allow a soldier to carry only one mine at a time. So after a soldier has discovered and picked up a mine, he will reach the border to clear the mine out of the area. During this period, he is able to detect another mine, but his pack is full at that time, thus he cannot pick another mine up. What he is able to do then is just inform the other roaming soldiers to come and clear the mine away. In this case, a message carrying the information will be sent indicating the position of the detected mine.

In the application, soldier agents can utilize ACL messages to exchange information. Basically five kinds of ACL templates are defined and utilized for these MSG soldier

Figure 6.5: Soldier behaviors in the MSG

agents:

- *informPosition*: tell other soldiers about his new position;
- *informPick*: announce that he has picked up a mine;
- *informDiscard*: announce that a task is completed;
- *informMine*: inform other soldiers about a detected mine;
- *informAck*: show acceptance of a task about a mine to the sender.

It is obvious that an *informMine* message is always sent in the soldier's busy state, and an *informAck* message is always sent in the soldier's available state.

6.4.3 Freehands Hash Table

A *freeHands* hash table is kept by every soldier, containing all the names and positions of those soldiers with free hands. The operations upon the *freeHands* are:

- if receive an *informDiscard*, add the sender's name into the *freeHands*;
- if receive an *informPick*, remove the sender's name from the *freeHands*;
- if receive an *informPosition* and the *freeHands* contains the sender's name, update the sender's position in the *freeHands*.

We can make good use of the freeHands hash table to achieve coordination between these soldiers. For example with this knowledge, every time a soldier in the busy state detects a mine, he will compare the distances between the mine and each soldier in his freeHands hash table. Only the soldier with the shortest distance to the mine will be notified. Then the receiver will use an *informAck* message indicating the acceptance of the task, and will start to walk towards the mine to clear it away.

It is important to understand that in practice there are some special cases, in which a notified soldier must transfer the task about clearing a mine to another soldier instead of starting to process it. They are, (1) if the soldier encounters another mine when processing a task, and (2) if the soldier receives another message about a task after he has accepted one. In the first case, he needs to forward the original task to another appropriate soldier. In the second case, the new task is forwarded. An alternative solution to this problem is to let the soldier take the task about the nearer mine and transfer the other one to another soldier. We should also notice that when a soldier's state is not identified yet, namely, when he is picking up a mine or discarding a mine, he might send an *informMine* message to himself. In this case, unnecessary message cycles occur. However, this problem can be solved by simply not sending *informMine* messages to the soldier himself.

6.5 Experimental Results

For a MAS, the most interesting variables include the number of agents in the system, the number of messages for a single agent, and the number of conversations in which a single agent is involved. By altering some of these variables, we have tested both the improvement of working efficiency of agents and the transmission time of ACL messages. The former is measured in simulation time, and the latter is measured in execution time in our experiments.

6.5.1 Working Efficiency

To evaluate the working efficiency of soldiers in the MSG, we use PCs configured with 2.20GHz Pentium 4 processor and 1.00GB RAM. The agent federate and the environment federate are on separate machines connected via the network. Two situations have been tested to compare the results. One is a large environment with a small quantity of mines distributed (1%), the other one is a comparatively small environment with a lot of mines distributed (10%). In Figure 6.6 and Figure 6.7, we can see that agent cooperation brings considerable improvement of the working efficiency. If only one soldier is working, he needs much more time to finish the game than the cases of multiple soldiers. A little increase in payout may gain much more reward.

Figure 6.6: Result with 1% mines distributed

For testing the interaction pattern of coordination, we apply the algorithms illustrated in Section 6.4 to the soldiers. When no communication exists, the multiple agents only work together without coordination. Comparing Figure 6.6 and Figure 6.7, we draw a conclusion that the coordination strategy is not always required in every situation. This

Figure 6.7: Result with 10% mines distributed

is because if mines are sparse, the rate of encountering another mine is small, and it is not likely that a busy soldier may inform another one about a newly discovered mine, thus the result does not differ much with or without communication. However, if mines are dense, the effect of communication emerges and it brings an attractive improvement of the efficiency with the same number of soldiers, as shown in Figure 6.7. It is interesting that the improvement in working efficiency is more noticeable if mines are clustered, because soldiers in coordination may come to the same area to remove the remaining mines if called for help by another. Examples of the environment with distributed mines and clustered mines are shown in Figure 6.8. The results of Figure 6.7 use clustered mines. No doubt this situation benefits a lot from MAS's communication.

In conclusion, the efficiency improvement brought by communication varies in different situations. In the case of a small quantity of mines distributed in a large environment, the main increase in agents' working efficiency is due to the number of agents. In the case of a large quantity of mines distributed in a small environment, the facility of communication

Figure 6.8: Snapshots of the environment with, (a) distributed mines; (b) clustered mines.

gives noticeable gains in agents' working efficiency.

6.5.2 Transmission Time of ACL Messages

We measured the execution times where a number of ACL messages are transmitted between agents in each time-step of the simulation. These are tested using three PCs with the same configuration as before connected via the network. As shown in Figure 6.9, one of these three machines — machine 3 builds up an environment shared by four agents — a , b , c and d , and the other two machines, machine 1 and machine 2, support these agents which are located in three different gateway federates — gateway federate 1, gateway federate 2 and gateway federate 3. Figure 6.10 shows the experimental results of three situations for one-way transmission, including sending ACLs to an agent located in the same federate (from agent a to agent b in Figure 6.9), sending ACLs to an agent located in a different federate but the same machine (from agent a to agent c in Figure 6.9), and sending ACLs to an agent located in a different federate and also a different physical machine (from agent a to agent d in Figure 6.9). Figure 6.11 shows the experimental results of two-way transmission for these three situations.

In both Figure 6.10 and Figure 6.11, the execution time does not change much when 1, 5 or 10 ACL messages per time-step are transmitted from an agent to another or

Figure 6.9: Four agents for testing messages' transmission time

in both directions at the same time. We find that these results remain stable and are very close to the situation when there is no message transmitted. However, when the number of messages reaches a large value, such as 50 and 100 messages per time-step, the difference among the execution time of the three situations becomes distinct. It should be noted that sending messages to an agent located in the same federate always keeps a stable value of execution time. This is because in this case there is no message that needs to be transmitted between federates. This avoids the need to convert messages to/from byte sequences, before the gateway federate can send them as interactions and after receiving them. The encoding and decoding are time-consuming processes that cost computing resources. This also explains why the time of sending messages to an agent in a different federate and also a different machine is less than the time of sending messages to an agent in a different federate but the same machine. The former does not increase as

Figure 6.10: ACL messages' transmission time (one-way)

Figure 6.11: ACL messages' transmission time (two-way)

much because the encoding and decoding can be carried out in parallel, whereas the latter requires the encoding and decoding to take place on the same machine. Figure 6.11 shows that sending messages two ways needs more execution time compared with the results in Figure 6.10, because in this case, twice as many messages need to be processed at the same time, being encoded and decoded. Moreover, for these results, a factor should also be taken into account that messages may encounter network delays when transmitted, however this is not crucial for a local area network.

In conclusion, the gateway federates and the mailbox agents work correctly in all the cases according to the conditions for consistency described in Section 6.1. We have found that the transmission of ACL messages is also efficient unless the number of messages sent by an agent is very large, which is not general in practice. Agents do not need to send messages in each time-step of the simulation. In general cases, they only communicate when necessary, thus sending ten or even more messages in one time-step is unlikely for an agent. However, those cases are still tested as upper bounds, and demonstrate that our mailbox agents and synchronization mechanisms are effective.

6.6 Summary

In this chapter, we have addressed the issues of how to enable the agent communication capabilities for a MAS that is integrated into an HLA-based distributed simulation. The focus is on keeping causality and consistency of the system. We have advanced the MSG prototype system into a system in which agent coordination is adopted based on different scenarios. A detailed strategy and algorithms have been presented with ACL templates defined. The idea of mailbox agent and two constraints have been applied in agent-to-agent communication in distributed simulations.

Experimental results have been given testing both the working efficiency of agents

and the transmission time of ACL messages. The agents' working efficiency has been measured and compared on the prototype system. Coordination among soldier agents has been shown to be effective, especially when target mines are dense and clustered. At the same time, we have demonstrated that using mailbox agents is an effective way of transferring ACL messages. In summary, in distributed agent-based simulations, a certain level of coordination also provides a larger viewpoint for every individual agent that has only partial information about the area. Communication overcomes the shortcoming of limited knowledge about the environment, and brings a better working efficiency for the agents compared to the situation where no ACL messages and coordination are applied.

Chapter 7

Model Resolution in MAS Simulations

When more and more agents are utilized in a distributed MAS simulation, different kinds of agents with different behaviors and functions are involved. Although they could have their own representations about the same phenomenon, some of these representations may be shared by other agents in the simulation. At the same time, agent hierarchies are very commonly applied in military simulations. Some agents (for example, a commander of a platoon) may be more powerful than the others and have control over them (for example, soldiers in a platoon). They will not pay attention to the same details as the latter but they influence the execution of these agents. In this chapter, we will address the issues of MAS simulations from the perspective of heterogeneous relationships among agents and their representations. Multi-resolution approaches that are relevant to agent simulation such as concurrent agent representations and aggregation-disaggregation will also be investigated.

7.1 Descriptions of MAS Hierarchies

In distributed simulations of MASs, because of the possibility of the existence of different agents with various behaviors, the relationships among the agents might be heterogeneous. We have explored three kinds of architectures for MAS hierarchies including horizontal layered MAS, vertical layered MAS and hybrid layered MAS. These will be illustrated in this section.

Wooldridge and Jennings categorize agents into reactive agents, deliberative agents and hybrid agents [64]. They present layered agent architectures for a single agent including horizontal layering and vertical layering [30]. The layers are arranged horizontally or vertically to distinguish the situations where all layers have access to the agent's sensors and effectors, and where only one layer has access to the agent's sensors and effectors. Our perspective of the classification of MAS hierarchies is not the same as theirs. Instead of discussing the agent architecture from the perspective of different layers of a single agent, we pay more attention to the case of MASs. In our approach, a MAS could be composed of a number of autonomous agents, each of them being a distinct agent that has both a reactive layer including perception and reaction interfaces, and a deliberative layer such as a reasoning system. In other words, each agent can have a full family of sense-deliberate-react functions which enable it to “perceive”, “think” and “respond”. It is possible to construct these agents freely into different hierarchies. This approach is flexible in the sense that agents can either behave independently or join together into groups hierarchically.

7.1.1 Horizontal Layered MAS

Figure 7.1 shows a tree-like hierarchy for multiple agents. The agents perceive and act in three different agent layers, thus their perception about the environment and the power

Figure 7.1: Horizontal layered MAS

of control may not be quite the same. For example, in the lowest agent layer 1, soldiers can see trees, detect mines and remove obstacles when marching. These soldiers have a commander who is possibly not at the front line. However he can control the soldiers and assign tasks to them, in a higher agent layer 2. The commander need not pay attention to the same details as the soldiers. What he cares about is which direction should the soldiers go, how many soldiers have died, how many of them are available, and what kind of mission should be carried out by the platoon next. Accordingly, he does not need too much detailed information such as when, where and who detects a mine, unless it is necessary. In an even higher agent layer, a general who is possibly sitting in the office away from the battlefield, can control these commanders. His information about the battlefield comes from the commanders and the commanders' information from the soldiers.

The characteristic of the horizontal layered MAS is that agents in a upper layer

perceive the physical environment indirectly¹, via the input from agents in lower layers. The agents in upper layers also have reactive components. However they are situated in an environment which is composed of agents, thus they deliberate according to the inputs from the lower layers and respond to those agents. Generally the agents in upper layers have the power to control the agents in lower layers in some way too. This architecture is suitable for the cases where autonomous agents have different degrees of power to control others.

The advantage of the horizontal layered MAS is that unnecessary information can be filtered out before submitting to an upper layer, and this may help the agents make a decision more quickly. By applying a horizontal layered MAS, a powerful agent can have a holistic view of the environment and a global control over some other agents, assigning tasks to them, and this in some sense overcomes their limitations of partial information about the environment.

7.1.2 Vertical Layered MAS

In the vertical layered MAS, agents are arranged in layers that are vertical to the input and output data flow from/to the physical environment. Different agents are allowed to perceive and act upon the environment in different ways, such as seeing, listening, attacking, and signaling. Their perception and reaction about the environment occur in the same vertical layer, no matter whether they are direct or not. Different kinds of agents may have different perspectives about the world and be in charge of different objects. In this case, more agents with various kinds of behaviors can be applied at the same time. This is very useful for the situation where more than one functionality is required to solve a problem.

¹There exist two possibilities for “embedded” agents. If they are directly situated in a simulated physical environment, their perception and reaction are called *direct*. Otherwise, if they are situated in an environment which is composed of agents, then their perception and reaction are called *indirect*.

Figure 7.2: Vertical layered MAS

For example, in Figure 7.2, a battlefield includes the land where soldiers fight, and the sky where pilots control fighter planes to attack. The soldiers on the ground can communicate with the pilots in the sky to exchange information about the battlefield, such as where there are many enemies and information about their weapons. Soldiers can detect enemies hidden in the jungle, fight with them and shoot. However they can also pass the information about their enemies to pilots and let them release bombs to attack. Pilots have a larger field of view than soldiers, thus they can detect the situation on the ground and report it by also sending messages to the soldiers. The soldiers and pilots have different specified functions and behaviors. This is a simple example of a vertical layered architecture of multiple agents.

The characteristic of the vertical layered MAS architecture is that the agents in the architecture can use different interfaces to perceive and react upon the environment. There is no control relationship among the agents in this architecture. The advantage of the vertical layered MAS is that agents embedded in the same environment are enabled

Figure 7.3: Hybrid layered MAS

to have different interfaces of perceptual input and action output, and to think over the world in different ways, thus the functionalities are made more distinct and specific for these agents. Accordingly agents are made more professional, and a big task such as attacking enemies could be divided into subtasks that include attacking on the ground using guns and attacking in the sky using fighter planes.

7.1.3 Hybrid Layered MAS

Sometimes, a MAS's architecture is not so simple as to be classified as horizontal layered or vertical layered. It is possible to have an architecture that is composed of both the above two kinds of architectures, which is called a hybrid layered architecture. For example, in Figure 7.3, there are three horizontal layers that contain different agents, layer $h1$, $h2$ and $h3$, in which agents jointly execute under the hierarchical control of the agents in higher layers. The layer $h1$ consists of two vertical layers, layer $h1 - v1$ and $h1 - v2$, in which agents perceive and act using different representation models.

This kind of architecture is more complicated. However it is more flexible and can meet some special requirements in some cases. It can have both the advantages of the horizontal layered MAS and the vertical layered MAS. Basically, the above three kinds of hierarchies can cover the commonly used models of MAS architectures.

7.2 Multi-Resolution Modeling

Multi-Resolution Modeling (MRM) is a way of representing objects at multiple levels of detail. This issue has been explored in a wide range of applications in a number of domains. It is concerned with resolving conceptual and representational differences arising from multiple levels of resolutions.

There are different approaches to MRM. However, no matter which approach is applied, MRM has been proven to be beneficial in many areas. For example, in multi-resolution graphical models, the system maintains multiple representations, or levels of detail, of an object and renders the appropriate representation depending on its distance from the viewer. Thus at run-time, users are permitted to change the level of detail [34], and an appropriate level can be selected for visually-appealing rendering [47]. In multi-level computer games, players can control characters inhabiting a world displayed at multiple resolutions. They may be interested in the most detailed resolution level to make an order or a lower resolution level to command a team. In hierarchical autonomous agent modeling, agents may jointly execute multiple layers in order to utilize the capabilities of each layer [62], for example, to use the deliberative layer to arrange the next behavior to complete a task, and to use the reactive layer to perceive and react to the environment.

7.2.1 High and Low Resolution Models

The concept of *model*² captures the semantics of a phenomenon in terms of some well-defined concepts. Simply, objects are called *entities* in a model. The *representation of an entity* is a way of describing the entity and its properties and processes³. The *representation of a model* is the union of the representations of the entities in the model. An *attribute* indicates an element of the representations of an entity. A *relationship* between two attributes describes how the value of an attribute changes according to changes in the value of another attribute.

When talking about representation of models, *resolution* is obviously a term that needs to be defined. Davis and Hillestad have made it easier to understand this multifaceted concept. In their description, resolution has various aspects including entity, attribute, logical dependency, process, spatial and temporal aspects [15]. To make the distinctions between high resolution and low resolution, a military example can be used:

- *Higher entity resolution* might mean modeling units as small as soldiers rather than platoons.
- *Higher attribute resolution* might mean modeling the number of different weapons held by each soldier in a platoon rather than the subtotal “strength” of the platoon.
- *Higher logical-dependency resolution* means including more constraints on the attributes and their inter-relationships, for example, the sum of the soldiers in the platoons comprising a company should equal the number of soldiers in the company.
- *Higher process resolution* might mean computing the cost at the platoon level, rather than computing the cost at the company level and then spreading it equally

²In the research area of simulation, a *model* is a representation of a real system, and a *simulation* is defined as an implementation of a model.

³Process is defined as a procedure: a particular course of action intended to achieve a result.

across the company's platoons.

- *Higher spatial and temporal resolution* means using finer scales for space and time.

For all the above aspects, *lower resolution* means the opposite. Basically, by using *low-resolution modeling*, users are enabled to make an initial cut at problems, comprehend the problem as a whole, reduce costs, speed up analysis, and make use of low-resolution knowledge to calibrate higher resolution models. By using *high-resolution modeling*, users can understand phenomena from the underlying perspectives, represent detailed knowledge, and make use of it to calibrate low-resolution models.

7.2.2 MRM Approaches

Davis and Hillestad introduce variable-resolution modeling (VRM) and investigate the related issue of developing integrated families of models [15]. *Variable-resolution modeling* means building new models or model families so that users can change readily the resolution at which phenomena are treated. This approach enables discrete changes in resolution within a single model or by moving from one model to another within a family, like zooming in/out. Moreover, *cross-resolution model connection* is defined as linking existing models with different resolutions, which is claimed to be a seamless design that permits changing resolution with both smooth consistency of representation and consistency of prediction.

In Natrajan's approach [50, 43], a Multi-Representation Entity (MRE) maintains *concurrent representations* which are defined as the representations of jointly-executing models. The representation of each model in a multi-model exists within a MRE at all times. Figure 7.4 shows such a multi-model — *MRE*, consisting of two models, $Model^A$ and $Model^B$. The MRE maintains Rep^A and Rep^B to describe objects of both $Model^A$ and $Model^B$. As we see, in Figure 7.5, P describes an object in $Model^A$, and Q_{1-3}

describe the same object in $Model^B$. E_1 is a MRE consisting of both the representations of P and Q_{1-3} , describing the same object at multiple resolution levels. E_1 is capable of interactions with both E_2 and E_3 which are entities that describe objects in $Model^A$ and $Model^B$ respectively. In this proposal, each MRE either maintains or efficiently furnishes the states at all desired resolution levels, and permits interactions at all resolution levels at all times. A survey of the approaches to multi-resolution modeling on DIS/HLA is given in [48].

Figure 7.4: Concurrent representations as proposed by Natrajan

Figure 7.5: Multi representation interaction as proposed by Natrajan

In this project, we also investigate the aggregation and disaggregation capabilities of the system which enable dynamic changes in agent resolution during the simulation execution. Simply, aggregation is defined as a transition from high resolution models to low

resolution models, and disaggregation is defined as a transition from low resolution models to high resolution models. This may be constructed with selected viewing, where the resolution is fixed. In selected viewing, the model is always executed at high resolution, but the user may select to view only those state corresponding to low resolution.

Both the approaches of selected viewing and aggregation-disaggregation only execute representations at one resolution level at any given time. However in some cases, both high-resolution models and low-resolution models may be needed at the same time. The technique of concurrent representations is valuable in solving this problem. We present a framework to represent objects at multiple levels of detail and to maintain the consistency of agents, that is discussed in the follows. In our approach, both the concurrent representations and aggregation-disaggregation are adopted, and we examine the tradeoff between these two solutions. Finally, an appropriate architecture called a Cross-Resolution Agent Architecture (CRAA) is constructed to meet these requirements. Chapter 8 gives more detailed discussions about this architecture.

7.3 Multi-Resolution Modeling in Agent Simulations

When multiple agents are used to represent entities in a simulation, they may have different representations to describe the same phenomenon at multiple resolution levels. Associated with the description in Section 7.2, we call an agent with representations at a high resolution level a high resolution agent (HRA), such as a soldier who keeps the representation of his own position, and we call an agent with representations at a low resolution level a low resolution agent (LRA), such as a commander of a platoon who keeps the representation of the whole platoon's position.

7.3.1 Concurrent Agent Representations

In Section 7.2.2, we introduced Natrajan's idea of concurrent representations. As shown in Figure 7.4 and Figure 7.5, in Natrajan's model, concurrent representations are defined as the representations of jointly-executing models within a multi-resolution entity (MRE). In his model, each MRE maintains the representations at all desired resolution levels, and a MRE permits interactions at all resolution levels at the same time. However, in a MAS, although there are multiple agents and these agents may have multiple representations, it is not necessary to maintain the representations at both high and low resolution levels in the same agent. The agents at a higher resolution level such as soldiers in a battlefield have more detailed representations about the environment than the agents at a lower resolution level such as the commander of a platoon. For reducing the consumed resources, it is also better that each agent only needs to maintain the representation(s) at its desired resolution level instead of all resolution levels. Thus Natrajan's model is not appropriate for our purposes in the area of MAS simulations.

In our proposal, there are entities with representations at different resolution levels such as HRAs and LRAs, and we also have jointly-executing models of representations. But the representations at different resolution levels are not in the same entity/agent. Each agent just needs to keep the representations of its hierarchical layer.

As shown in Figure 7.6, we define *concurrent agent representations* as jointly-executing models representing attributes of the same phenomenon in the horizontal layered MAS⁴. The set of representations Rep^L are held in the LRA at a low resolution level but a high agent layer. In all the HRAs under this LRA such as HRA_1 and HRA_2 , the sets of representations including Rep^{H_1} and Rep^{H_2} are held at a high resolution level but a low

⁴In this model, we only consider the situations where a HRA has no more than one LRA that has control over it. Otherwise, different LRAs within the same agent layer should interact with each other first, before making a decision for the HRAs of the lower agent layer.

agent layer. Here Rep^L , Rep^{H_1} and Rep^{H_2} are all sets of representations because each agent has more than one attribute to be represented. For example, for the horizontal layered agents presented in Figure 7.1 in Section 7.1.1, the general agent keeps the representations in the company layer (Layer 3), the commanders keep the representations in the platoon layer (Layer 2), and the soldiers keep the representations in the soldier layer (Layer 1). In this case, each of them need not maintain the representations at all three resolution levels. Concurrent agent representations can also be found in the hybrid layered architecture which includes the horizontal layered architecture.

Figure 7.6: Concurrent agent representations

There are two classes of concurrent representations involved in the domain of MASs, namely, concurrent representations of agent's external and internal attributes:

- *Agent external attributes* are an agent's perceptual inputs about the environment, for example, there is a mine beside this agent, a river is on the left side of the agent. They are usually retrieved from the environment via the agent's sensor.
- *Agent internal attributes* indicate an agent's current behaviors, activities and capabilities, such as the agent is approaching borders, how many mines are inside his pack, and he is busy or not. They are usually saved in the agent's knowledge base.

For the above two classes, indirect perception such as representations received from other agents can be classified using the same way. An external attribute can influence an internal one, for example, if an enemy emerges (external attribute), a soldier will start to attack turning the state from idle to busy (internal attribute). An internal attribute can also decide the agent's next action which will influence external attributes, for example, in the MSG prototype, if the soldier's hands are free (internal attribute), he will continue to roam and detect mines, otherwise, he will approach the border to discard the mine. Thus the surrounding information retrieved from the environment (external attribute) will not be the same.

By capturing the key property of representations and their relationships, we can define a simulation of one agent simply as a tuple of attribute representations, relationships and simulation times:

$$Simulation = \langle Rep, Rel, T \rangle$$

Rep is the set of representations of one agent's attributes including both the internal attributes and external attributes. We assume that $Rep \neq \emptyset$. This indicates that representations exist for a simulation model. Rel is the set of all relationships within one agent in the simulation. Here the term relationship indicates the association among agent attributes. For two attributes p and q , such that $p \subseteq Rep \wedge q \subseteq Rep$, a relationship $r \in Rel$ is a mapping $r : p \rightarrow q$. For example, in the MSG, the changes in a soldier's position may influence his surrounding information.

When multi-resolution modeling is involved, consider a situation in which the MAS simulation consists of two kinds of hierarchical agents, for example, LRA and HRA that are in different horizontal layers as described in Section 7.1.1. LRA may be the commander of a platoon, and HRA may be a soldier of this platoon. Then a multi-resolution simulation $Simulation^M$ constructed from them is defined as:

$$Simulation^M = \langle Rep^M, Rel^M, T^M \rangle$$

$$Rep^M = Rep^L \cup Rep^H$$

$$Rep^H = Rep^{H_1} \cup Rep^{H_2} \cup \dots \cup Rep^{H_i}$$

$$Rel^M = Rel^L \cup Rel^H \cup Rel^{cross}$$

In these formulas, Rep^M is a set of what we call concurrent agent representations. It can be constructed by including all of the representations at a low resolution level, Rep^L , and the representations at a high resolution level, Rep^H . Rep^H is composed of the representations of all the HRAs under an LRA, including Rep^{H_1} of HRA_1 , Rep^{H_2} of HRA_2 , etc. For any attribute a , $a \in Rep^L \vee a \in Rep^H \equiv a \in Rep^M$. A cross-agent relationship $r \in Rel^{cross}$ is a mapping $r : p \rightarrow q$, such that $p \subseteq Rep^L \wedge q \subseteq Rep^H \vee q \subseteq Rep^L \wedge p \subseteq Rep^H$. For a relationship r , $r \in Rel^L \vee r \in Rel^H \vee r \in Rel^{cross} \equiv r \in Rel^M$.

In this multiple representation model, the time-steps of Rep^L and Rep^H should be compatible, which means that if T^L , T^H and T^M are the sequences of times associated with Rep^L , Rep^H and Rep^M respectively, then Rep^L and Rep^H are defined for all times in T^M . T^M is a subset of the union of T^L and T^H , so that $T^M \subseteq T^L \cup T^H$.

7.3.2 Agent Resolution Transition

As discussed in Section 7.1, when multiple agents are involved in a simulation, different hierarchies may be adopted. In agent simulations, dynamic changes of the agents' model of existence are necessary in some cases. For example, one of the common soldier agents may be promoted to a commander or a leader (change in horizontal layered architecture), or an agent which represents a whole platoon of soldiers may break into separate agents that represent individual soldiers (change from an LRA to HRAs). This is especially

true in military simulations where an army corps can break down into more and more detailed subdivisions hierarchically. It will result in changes of agent resolution such as *agent aggregation* and *agent disaggregation*.

Agent aggregation/disaggregation transition is closely related to the issues of dynamic changes in resolution. *Aggregation* is defined as a transition from high resolution models to low resolution models, and *disaggregation* is the converse transition process, which is from low resolution models to high resolution models. Aggregation and disaggregation relationships correspond to the logical relationships “has a” or “is a component of” respectively. Accordingly, in this thesis, *agent disaggregation* is defined as a transition from low-resolution agents (LRAs) to high-resolution agents (HRAs), and *agent aggregation* is the converse transition process, which is from HRAs to LRAs.

Figure 7.7 gives an example of agent aggregation. Originally soldiers in a platoon are represented by distinct software entities. Before the aggregation, they have individual attributes including *positions*. After the aggregation, none of them will keep private attributes, but only a shared *position*, say (x, y) . The platoon will not keep the detailed information of all these soldiers any more. From then on, only low resolution representations are utilized.

Figure 7.7: Agent aggregation

Figure 7.8 gives an example of agent disaggregation. Originally, there is just one agent that represents a platoon. It is a platoon that consists of several soldiers that behave as a whole. This agent has internal attributes of *position*, say (x, y) . After the disaggregation,

tion, soldiers become distinct software entities that have *position* attributes respectively, say $(x_1, y_1), (x_2, y_2) \dots (x_n, y_n)$. Arbitrary values of (x_i, y_i) may be chosen, however the average value of their *position* must be equal to the position before disaggregation. Each of them may make its own decision about the next behavior.

Figure 7.8: Agent disaggregation

Before the trigger of the disaggregation such as the assignment of tasks, soldiers in the platoon do not need to perceive and make decisions one by one. They just need to move together. In the situation after the disaggregation, full disaggregation and partial disaggregation are distinguished. If the trigger makes all entities disaggregate from the platoon, it is called a *full disaggregation*. If only some of the entities disaggregate from the platoon, it is called a *partial disaggregation*. The latter avoids unnecessary entire disaggregation that may occur in the former. For example, if a team discovers a mine and decides to clear it away, just one or two soldiers need to disaggregate to accomplish this task. The remaining soldiers may still be in the group, moving together and acting as a whole.

7.4 Maintaining Representation Consistency

In our simulation models, maintaining representation consistency ensures that during the simulation execution, any pair of agent attributes will not have values that conflict with one another no matter whether the attributes are within the same agent or in different agents that execute at different resolution levels. Inconsistencies may arise if agents

with representations in different resolution levels have attributes that represent the same phenomenon, and they execute concurrently. Inconsistencies may also arise when agents' representation resolution changes, such as when agents aggregate (HRAs $\rightarrow$ LRAs) and disaggregate (LRAs $\rightarrow$ HRAs).

For an architecture which supports hierarchical agents and dynamic changes in their representation resolution in MAS simulations, we need to solve some problems such as how to ensure the consistency of concurrent representations of agents' attributes, and how to ensure the consistency in the agent resolution transition during the simulation.

7.4.1 Attribute Dependency Graph and Mapping Function

It is crucial to maintain the consistency of agent attributes among the hierarchical agents along the increasing direction of the time axis. The technologies of attribute dependency graph (ADG) [43], and mapping function can be utilized to meet the requirements of maintaining consistency among attributes of the agent at different resolution levels.

In the simple ADG shown as in Figure 7.9, the left node corresponds to an agent attribute a , and the right node corresponds to another agent attribute b . They may be attributes within the same agent, and for multi-resolution models, they could also be in different agents as shown in Figure 7.6. The arrow connecting a and b indicates that b depends on a , or a affects b . So if the value of a changes, the value of b may also change. If the value of b changes, there is no requirement for the value of a to change. For the relationship in this figure, a is the independent attribute and b is the dependent attribute.

Figure 7.9: A simple ADG

In our multi-resolution simulation model $Simulation^M$, an ADG can capture de-

dependencies among representations of agent attributes, and application-specific mapping functions can be used to translate these attributes. In the ADG, every node corresponds to an agent attribute in the Rep^M of a multi-resolution simulation, and every arrow corresponds to a dependency among these agent attributes, which is represented by an arrow between two agent attributes. Each arrow is a relationship among agent attributes $r \in Rel^M$ associated with a mapping function that translates the changes in one agent attribute to changes in another agent attribute. The ADG in Figure 7.9 does not show how b must change when a changes, however application-specific mapping functions could encode this.

7.4.2 Consistency of Concurrent Agent Representations

Figure 7.10 shows the definition of concurrent agent representation consistency. $S(t_0, t_1)$ indicates the state transition of LRA from time t_0 to time t_1 , and $s(t_0, t_1)$ indicates the state transition of HRA from time t_0 to time t_1 . Between HRA and LRA, there exist two direction mapping functions, f for mapping low resolution representations of the LRA to associated high resolution representations of the HRA, and g for mapping high resolution representations of the HRA to associated low resolution representations of the LRA, i.e., $LRA(t) = g(HRA(t))$ and $HRA(t) = f(LRA(t))$ at time t . Maintaining consistency among concurrent agent representations must ensure that:

$$S(t_0, t_1) \circ g = g \circ s(t_0, t_1)$$

$$s(t_0, t_1) \circ f = f \circ S(t_0, t_1)$$

These two only guarantee *partial consistency* of the system. If f and g are both 1 – 1 and onto mappings, we can get $f \circ g = g \circ f = identity$. Accordingly, we can derive:

$$f \circ S(t_0, t_1) \circ g = s(t_0, t_1)$$

$$g \circ s(t_0, t_1) \circ f = S(t_0, t_1)$$

These two formulas ensure the form of consistency defined by Davis, which states that if we can obtain the correct detailed state by moving clockwise or counterclockwise around the diagram, then we say there is *complete consistency* [15].

Figure 7.10: Concurrent agent representation consistency

7.4.3 Consistency of Agent Resolution Transition

In Section 7.3.2, we have talked about agents' dynamic changes in resolution. Obviously, during the execution of a simulation, agents' attributes should be consistent before and after the aggregation/disaggregation. For example, in the MSG, the states of newly separated soldiers of a platoon need to be initialized during the disaggregation. Before the disaggregation, all soldiers are available to pick up mines. After encountering a mine, they disaggregate. Only one of the soldiers' states will change to busy, and the others remain available for picking up mines. After the disaggregation, it is unacceptable if all of the soldiers remain free, or all of them become busy, because only one mine needs to be picked up in this case.

Figure 7.11 shows the definition of agent resolution transition consistency. $S(t_0, t_1)$

indicates the state transition of LRAs from time t_0 to time t_1 , and $s(t_0, t_1)$ indicates the state transition of HRAs from time t_0 to time t_1 . Let agg indicate the aggregation operation from the HRA to LRA and dis indicate the disaggregation operation from the LRA to HRA, such that $LRA(t) = agg(HRA(t))$, and $HRA(t) = dis(LRA(t))$ at time t . In aggregation/disaggregation, generally there only exists one state at any given time, namely, aggregated state or disaggregated state. So just one of the time transitions will hold at a given time, S or s . It is possible to calculate one of them from the other one, such as:

$$s(t_0, t_1) = dis \circ S(t_0, t_1) \circ agg$$

$$S(t_0, t_1) = agg \circ s(t_0, t_1) \circ dis$$

However, these formulas do not guarantee consistency. For agent simulations, agent attributes should be specified and kept consistent during agent resolution transition. Problems that can arise during agent aggregation and disaggregation are discussed further in Section 8.3.3.

Figure 7.11: Agent resolution transition consistency

7.4.4 Enforcing Consistency

For both concurrent agent representations and agent resolution transition we need some mechanism for maintaining consistency. As shown in Figure 7.12, consistency enforcers are used to maintain consistency by capturing the cross-agent relationships of agent attributes. Each consistency enforcer takes charge of an agent attribute.

Figure 7.12: Consistency enforcer

Consistency enforcers use the ADG that captures dependencies among agent attributes. According to various requirements of particular simulations, application-specific mapping functions are used to translate changes in one agent attribute to changes in another. When the value of an agent attributes changes, the consistency enforcer determines how the value of another agent attribute must change. The consistency enforcer performs the changes by invoking the mapping functions.

The consistency enforcer maintains the consistency of representations during the simulation execution and is implemented in the agent code. For concurrent agent repre-

sentations, the consistency enforcer is executed as part of the agent simulation cycle. When inconsistency arises among a pair of agent attributes that represent the same phenomenon at different resolution levels, the consistency enforcer determines which value is correct and ensures that they will not have values that conflict with each other. For agent resolution transition, the consistency enforcer is executed to ensure the consistency every time there is an agent resolution transition such as aggregation/disaggregation.

In the next section, we will show how to construct an ADG and mapping functions for a commander-soldiers scenario. In Chapter 8, we will also introduce a cross-resolution agent architecture, which has both the capabilities of concurrent representations and consistent resolution transition.

7.5 Applying the ADG

As mentioned in Section 7.4, for maintaining consistency, the technique of ADG can be applied to capture cause-effect dependencies among agents' representations of attributes and states, Rel^M . In this section, we give a description of how to apply ADG and mapping functions in our proposed agent architecture.

The first step in constructing the ADG is assigning nodes to agent attributes in the simulation Rep^M . For example, as shown in Figure 7.13, two agent layers may be represented in a commander-soldier (CS) scenario: the platoon layer including a commander, and the soldier layer including two soldiers. Thus in this scenario,

$$Rep^M = Rep^{commander} \cup Rep^{soldier}$$

$$Rep^{soldier} = Rep^{soldier_1} \cup Rep^{soldier_2}$$

The commander agent who is in the platoon layer may have attributes for *Position* which indicates the position of the platoon, *Capability* indicating how many more mines

Figure 7.13: A commander-soldiers scenario

can be carried by the platoon in total, *State* indicating whether the platoon is busy or not, and *AroundInfo* indicating the surrounding information of the platoon. Here, the commander has agent attributes including *State*, *Capability*, *Position* and *AroundInfo*. Soldier agents also have these agent attributes indicating their positions, how many more mines can be carried, states and surrounding information, say, *State_i*, *Capability_i*, *Position_i* and *AroundInfo_i*.

$$Rep^{commander} = \{Position, Capability, State, AroundInfo\}$$

$$Rep^{soldier_i} = \{Position_i, Capability_i, State_i, AroundInfo_i\}, (i = 1, 2)$$

Table 7.1 and Table 7.2 give the definitions of the attributes in this scenario.

The second step is assigning arrows to dependencies among these nodes to capture the relationships in Rel^M .

$$Rel^M = Rel^{commander} \cup Rel^{soldier} \cup Rel^{cross}$$

Commander's Attributes	Type	Sample Values
<i>Position</i>	int[2]	{5,6}, {9,9}
<i>AroundInfo</i>	char[2][3][3]	{{{'s',' ',' '},{', 's','m'}},{', ' ','t'}}, {{', ' ',' '},{', 's',' '},{', ' ',' '}} (‘s’ = soldier, ‘m’ = mine, ‘t’ = tree)
<i>State</i>	boolean	true(busy), false(available)
<i>Capability</i>	int	1, 2, 5

Table 7.1: The definitions of the commander's attributes

Agent Attributes	Type	Sample Values
<i>Position</i> ₁ , <i>Position</i> ₂	int[2]	{5,6}, {9,9}
<i>AroundInfo</i> ₁ , <i>AroundInfo</i> ₂	char[3][3]	{{{'s',' ',' '},{', 's','m'}},{', ' ','t'}}, (‘s’ = soldier, ‘m’ = mine, ‘t’ = tree)
<i>State</i> ₁ , <i>State</i> ₂	boolean	true(busy), false(available)
<i>Capability</i> ₁ , <i>Capability</i> ₂	int	1, 2, 5

Table 7.2: The definitions of soldiers' attributes

Mapping functions can be assigned to these dependencies, which enables the construction of appropriate relationships for them.

In Figure 7.14, we show dependencies for the commander-soldier scenario. The arrows inside the box of the platoon-layer representations are the relationships included in $Rel^{commander}$. For example, the platoon's *Position* determines the platoon's *AroundInfo* which is the surrounding information that covers a specified range of area. Similarly, the arrows inside the box of soldier-layer representations are the relationships included in $Rel^{soldier}$. The arrows that cross the above two boxes are exactly those included in Rel^{cross} . For example, the positions of the soldiers influence the position of the platoon closely. The mapping functions are enumerated in Table 7.3.

The *commander's* attribute *Capability* which indicates how many more mines can be carried by the whole platoon has relationships with the soldiers' *Capability_i*. *Capability₁* indicates how many more mines can be carried by *soldier₁*, and *Capability₂* indicates how many more mines can be carried by *soldier₂*.

Consider a scenario where each soldier can carry two mines. At time t for these

Related Attributes	Mapping Function	Type
$Position,$ $AroundInfo$	$AroundInfo = range(Position)$ (A platoon's surrounding information depends on the platoon's position.)	$Rel^{commander}$
$AroundInfo,$ $State$	$State = (mineNum(AroundInfo) \geq Capability)$ (A platoon is busy or not depending on how many mines are detected around.)	$Rel^{commander}$
$Capability,$ $State$	$State = (mineNum(AroundInfo) \geq Capability)$ (A platoon is busy or not depending on how much capability is left.)	$Rel^{commander}$
$Position = (x, y),$ $Position_1 = (x_1, y_1),$ $Position_2 = (x_2, y_2)$	$x = (x_1 + x_2)/2,$ $y = (y_1 + y_2)/2$	Rel^{cross}
$AroundInfo,$ $AroundInfo_1,$ $AroundInfo_2$	$AroundInfo = AroundInfo_1 \cup AroundInfo_2$	Rel^{cross}
$State,$ $State_1,$ $State_2$	$State = State_1 \wedge State_2$	Rel^{cross}
$Capability,$ $Capability_1,$ $Capability_2$	$Capability = Capability_1 + Capability_2$	Rel^{cross}
$Position_i,$ $AroundInfo_i$	$AroundInfo_i = sensor(Position_i)$ (A soldier's surrounding information depends on the soldier's position.)	$Rel^{soldier}$
$AroundInfo_i,$ $State_i$	$State_i = (mineNum(AroundInfo_i) \geq Capability_i)$ (A soldier's state depends on how many mines are detected around him.)	$Rel^{soldier}$
$Capability_i,$ $State_i$	$State_i = (mineNum(AroundInfo_i) \geq Capability_i)$ (A soldier's state depends on whether he can carry any more.)	$Rel^{soldier}$

Table 7.3: Mapping functions in the commander-soldiers scenario

Figure 7.14: Dependency graph for the commander-soldiers scenario

two soldiers, $Capability_1(t) = 2$, and $Capability_2(t) = 2$. Then for the whole platoon, $Capability(t) = 4$. If these soldiers detect three mines as shown in Figure 7.15, each of them will try to pick up two mines. Since $mineNum(AroundInfo_1(t)) = 2$ and $mineNum(AroundInfo_2(t)) = 2$, according to Table 7.3, both soldiers are busy at that time. But for the platoon, $mineNum(AroundInfo(t)) = 3$, and it is smaller than the platoon's $Capability(t)$. Although all of the soldiers are busy, it is still a problem whether the platoon is busy or not at that time. Table 7.4 shows the relationships.

M	S	M	S	M

Figure 7.15: Two soldiers and three mines

As shown in Table 7.5, a consistency enforcer is applied here to ensure the consistency of *State*. The theory about the ADG and mapping functions is applied. According to Figure 7.14 and Table 7.3, the consistency enforcer in the platoon layer will determine

<i>Concurrent Agent Representations</i>		Rel^{cross}	Consistent?
Platoon Layer	Soldier Layer		
$Capability(t) = 4$	$Capability_1(t) = 2$ $Capability_2(t) = 2$	$Capability =$ $Capability_1 + Capability_2$	yes
$mineNum(t) = 3$	$mineNum_1(t) = 2$ $mineNum_2(t) = 2$	$AroundInfo =$ $AroundInfo_1 \cup AroundInfo_2$	yes
$State(t) = false$	$State_1(t) = true$ $State_2(t) = true$	$State = State_1 \wedge State_2$	no

Table 7.4: Attribute values at time t

how the value of $State$ must change when the values of $State_1$ and $State_2$ change, so that $State(t) = State_1(t) \wedge State_2(t) = true$. The consistency enforcer determines that the platoon should also be busy at time t , thus ensuring the values of $State$ are consistent with each other.

<i>Concurrent Agent Representations</i>		Rel^{cross}	Consistency Enforcer
Platoon Layer	Soldier Layer		
$Capability(t) = 4$	$Capability_1(t) = 2$ $Capability_2(t) = 2$	$Capability =$ $Capability_1 + Capability_2$	ok
$mineNum(t) = 3$	$mineNum_1(t) = 2$ $mineNum_2(t) = 2$	$AroundInfo =$ $AroundInfo_1 \cup AroundInfo_2$	ok
$State(t) = false$	$State_1(t) = true$ $State_2(t) = true$	$State = State_1 \wedge State_2$	$State(t) = true$

Table 7.5: Attribute values at time t using consistency enforcer

7.6 Summary

In MAS simulations, different kinds of agents with different behaviors and representations are involved. In this chapter, different kinds of MAS hierarchies have been classified. Moreover, different approaches of multi-resolution modeling have been introduced. We have investigated the problems of multi-resolution in the area of MAS simulations. Our approach to concurrent agent representations has been presented to meet the requirements in discussing hierarchical agents with representations at different resolution levels. Agent resolution transition such as aggregation and disaggregation has also been introduced.

We have addressed the issues about how to maintain the consistency of both concurrent agent representations and agent resolution transition. An attribute dependency graph and mapping functions have been used to maintain representation consistency of agents.

Chapter 8

Cross-Resolution Agent Architecture

In this chapter, we continue our investigation of multi-resolution modeling, especially in the area of distributed MAS simulations. We introduce a novel agent architecture, called a cross-resolution agent architecture (CRAA), which is an implementation of the idea of concurrent agent representations where agents with representations at different resolution levels are linked together. The CRAA involves building a hierarchy of agents across two or more agent layers. It can also support dynamic resolution transition of agents without the violation of consistency.

8.1 Atomic Agent and Mass Agent

Before describing the CRAA, it is necessary to introduce the terms *atomic agent* and *mass agent*. An agent can be implemented as a distinct software entity that executes individually. We call this kind of agent an atomic agent. In Section 7.1, we have addressed the issue of agent hierarchies from the perspective of relationships and control among these agents. In these agent hierarchies, we can use the term *peer agents* to name the agents in the same horizontal layer. If a peer agent is also implemented as an atomic agent, then we call it a *peer atomic agent*.

Peer agents are probably the most commonly applied agents in practice. Peer agents

sense from and react to the environment using the same perception input and action output interfaces, which means their representations of the environment are in the same format. They use the same way to perceive and act upon the environment, e.g. to detect an object at position (x, y) and to destroy/move an object. In this case, when these agents communicate with each other, it should not be necessary to translate the message before another one reads and understands it. Although they are different agents, they can easily understand each other. However, they may have different decision-making rules and behavior scheduling. For example, in Figure 8.1, the soldiers who are clearing mines from the battlefield are peer agents to the enemies who are planting mines in the battlefield. The commander is not a peer agent to either the soldiers or enemies.

Figure 8.1: Peer agents

For saving computing resources, it is not always necessary to set up an atomic agent for each intelligent entity. It is possible to create an agent to represent a group of intelligent entities at the same time. In the example above, if there are one hundred soldiers, since their representations about the environment use the same set of interfaces, and they have

the same decision-making rules, it is not necessary to create one hundred atomic agents respectively. We can implement just one software entity to represent all the soldiers. Inside this software entity, only one set of behavior scheduling is required, and all agents share the same decision-making rules. We call this kind of software entity a *mass agent* to distinguish it from atomic agents.

The preconditions for multiple agents to be wrapped into a mass agent are:

- they are peer agents, and
- they share the same set of decision-making rules.

For example, in Figure 8.1, only the soldiers in the box can be wrapped into a mass agent. Although the enemies are peer agents to the soldiers, they cannot be wrapped into the same mass agent, because the second precondition is violated. Figure 8.2 shows the resulting structure of a mass agent, where a group of soldiers are wrapped in the mass agent. Accordingly, a mass agent represents peer agents that share the same set of decision-making rules, and it supports concurrent execution of agent behaviors. The concurrent execution of agent behaviors can be implemented as parallel behaviors or sub-behaviors of an agent as described in Section 5.2. This is not the same as aggregation, as the soldiers are still represented at a high resolution level and can perform different actions.

8.2 CRAA Algorithms

In military simulations, we can use the mass agent to represent a team or a platoon. It is also beneficial if the mass agent could have control over all the agents inside it, synchronizing their decisions and even adjusting the behaviors, in order to reduce/prevent duplicated actions and mutually exclusive interactions among them. This is achieved

Figure 8.2: A mass agent representing three soldiers

using an agent architecture called a cross-resolution agent architecture (CRAA), which supports the idea of concurrent agent representations associated with the previous discussions in Section 7.3.1. The CRAA can be regarded as an extension of the mass agent architecture, which is enhanced to support agents in different hierarchical layers. Because of the central control of the CRAA, the decisions about agents' actions can easily achieve an agreement. Since agent aggregation/disaggregation is also usually required for military simulations, the issue about dynamic changes of agent resolution is also addressed in this architecture.

Figure 8.3 shows the cross-resolution agent architecture for two agent layers including an LRA and several HRAs. In this architecture, the agent with representations at the low resolution level and the agents with representations at the high resolution level jointly-execute with each other. Different agents could have concurrent representations of the same phenomenon. The LRA has the ability to create and subsequently delete an HRA if the deliberation over high resolution representations is necessary or more HRAs are required, and it always has overall control of all the HRAs under it. On the other hand, in cases where the deliberation over low resolution representations is not necessary, then only the HRAs will be applied without the activation of the LRA's overall-deliberation

and behavior-adjustment functions.

Figure 8.3: Cross-resolution agent architecture

The CRAA meets the requirement of supporting concurrent agent representations, and it also makes it possible to switch between high and low resolution agents. This architecture differs from previous architectures for multi-resolution modeling discussed in Section 7.2.2. It is not like a selected viewing model, in which the resolution is fixed. Neither is it simply an aggregation-disaggregation model, in which dynamic changes of resolution level are allowed, but only one resolution level is applied at a time.

8.2.1 Two Agent Layers

Considering the discussions in Section 7.4, we apply the following algorithm for the sense-deliberate-react cycle within the CRAA:

CRAA-algorithm

```

initialize-states();
loop
  get-new-input();
  begin LRA
    LRA-external = low-resolution-representation-generator(input);
    for all HRA
      begin HRA
        HRA-external = high-resolution-representation-generator(input);
        HRA-internal = HRA-deliberate(HRA-internal, HRA-external);
      end HRA
    LRA-external = calibrate(HRA-external);
  end for
  LRA-internal = LRA-deliberate(LRA-internal, LRA-external);
  for all HRA
    HRA-internal = adjust-behaviors(LRA-internal);
  end for
end LRA
  enforce-consistency();
  execute-actions();
end loop

```

There is only one LRA in this algorithm as generally there will be a separate CRAA agent for each LRA and the HRAs under it. First, all the states of the agents will be initialized including those of the LRA and the HRAs. The LRA and the HRA(s) can get the same input information from the environment. However, they have different representation generators to generate corresponding perceptions for the agents at different resolution levels. For the LRA, we call it a *low resolution representation generator*, which generates low resolution representations, and for the HRA, we call it a *high resolution representation generator*, which generates high resolution representations. After all HRAs

have finished their deliberation, the LRA processes its deliberation, and then adjusts the HRA(s)' behaviors according to the overall deliberation if necessary.

The adjustment of both HRA(s) and the LRA may be required in each cycle of the execution of the CRAA. Generally we only allow the LRA to change the behaviors of the HRA(s) under it, using the operation of *adjust-behaviors*. However, since the attributes of HRA(s) are usually more detailed, in the case where an LRA's attribute value is not accurate enough to meet some requirements, it is also possible for an HRA to alter the external attributes of an LRA, using the operation of *calibrate*. Consistency enforcers are applied at the end of each simulation cycle to maintain the consistency of concurrent agent representations.

We can see that in this algorithm, both concurrent representations of agent attributes and concurrent execution of agent behaviors are supported at different resolution levels. The architecture can also support aggregation/disaggregation. For example, HRAs such as soldiers are implemented as sub-behaviors of the extended mass agent that represents the platoon¹. During the aggregation/disaggregation phase, HRA(s) can be deleted/created by the commander of the platoon. Consistency enforcers are applied during these phases to ensure the consistency of agent resolution transition.

8.2.2 Multiple Agent Layers

The idea of the CRAA can be extended and applied in the situation where more than two layers of agents are involved in the simulation. The agent layers are numbered from the lowest to the highest agent layers in the horizontal agent hierarchy as shown in Figure 7.1. For example, agents with representations at high, middle and low resolution levels such as a soldier, a commander and a general, will be indicated by A_1 , A_2 , and A_3 respectively. Again, there is generally only one LRA in the top layer i indicated by A_i . Then the

¹Here, the platoon is the combination of a commander agent (LRA) and several soldier agents (HRAs).

algorithm applied with multiple agent layers is as follows:

CRAA-algorithm (with i layers)

```

initialize-states();
loop
  get-new-input();
  call CRAA-execute( $A_i$ );
  enforce-consistency();
  execute-actions();
end loop

```

CRAA-execute(A_i)

```

begin  $A_i$ 
   $A_i$ -external =  $A_i$ -representation-generator(input);
  for all  $A_{i-1}$  that are children of  $A_i$ 
    call CRAA-execute( $A_{i-1}$ );
     $A_i$ -external = calibrate( $A_{i-1}$ -external);
  end for
   $A_i$ -internal =  $A_i$ -deliberate( $A_i$ -internal,  $A_i$ -external);
  for all  $A_{i-1}$  that are children of  $A_i$ 
     $A_{i-1}$ -internal = adjust-behaviors( $A_i$ -internal);
  end for
end  $A_i$ 

```

Similar to the algorithm for two agent layers, both concurrent representations of agent attributes and concurrent execution of agent behaviors are supported in this algorithm. In this architecture, the inputs for the hierarchical agents are the same. However they need different generators to produce their external attributes according to the requirements of their resolution. Mapping functions are used to translate the changes in one agent attribute to changes in related agent attributes. For ensuring the consistency of agent attributes, consistency enforcers are applied during the simulation execution.

In conclusion, the CRAA algorithm is flexible enough to support agents with representations at different resolution levels. This architecture can be regarded as an extended form of mass agent with the additional capabilities of supporting concurrent agent representations and agent resolution transition.

8.3 Applying the Mass Agent and CRAA

8.3.1 Mass Agent Architecture

In the MSG prototype system, a mass agent can be utilized to represent a platoon that consists of soldiers that are members of the platoon. Figure 8.4 shows the architecture of a mass agent implementation which contains several soldiers with sense-deliberate-react cycles respectively. The implementation has a series of complex behaviors, composed of several sub-behaviors, with each of the soldiers implemented as a parallel behavior of the agent. Thus every soldier has a distinct behavior for the deliberation phase, which is processed concurrently with other soldier's.

Figure 8.4: The mass agent for MSG

To achieve the mass agent architecture, in practice we need to reorganize the behavior scheduling for each soldier into three phases, sense→ deliberate→ react, and enable them to loop together. Agents perform actions upon the environment as responses, and may transfer from one behavior to another in an agent simulation cycle.

Referring to Figure 5.2, we define the soldier behaviors as follows:

- wander: the soldier roams and detects mines.
- pick: the soldier picks up a mine.
- border: the soldier reaches the border.
- discard: the soldier discards the mine.

There are three attributes concerning the soldier's decision-making about the next action including *State*, *anyMine* and *anyBorder*.

1. State: is the soldier busy or available?
2. anyMine: is there any mine beside the soldier?
3. anyBorder: does the soldier stand beside the border?

Since the *anyMine* and *anyBorder* attributes are part of the *AroundInfo*, they are similar to the agent attributes of the example given in Section 7.5.

The soldier agents are executing as finite state machines. An agent's current state covers both its internal and external attributes including the environment information from the sensor and the messages from other agents. These together determine the agent's actions as shown in table 8.1. In Figure 8.5, we enumerate all the three-phase alternatives for a soldier's behavior scheduling compared to the previous behavior scheduling shown in Figure 5.15.

Internal Attribute handFull	External Attributes anyMine anyBorder		Behavior
false	false	false	wander
false	false	true	wander
false	true	false	pick
false	true	true	pick
true	false	false	border
true	false	true	discard
true	true	false	border
true	true	true	discard

Table 8.1: The internal and external attributes determine the soldier's behaviors

Figure 8.5: Three-phase behavior scheduling for soldiers

So far, in this scenario, all the agents are still in the same hierarchical layer. The simulation cycle is as follows, first of all, the mass agent gets the sensors via the object-to-agent communication channel from the gateway federate. Then it processes the sensors and matches them respectively for each soldier. The soldiers deliberate according to their attributes and then decide the actions as responses. After all the soldiers have finished this deliberation phase, the mass agent fetches every soldier's decision with his effector and passes the information back to the gateway federate. This is the common case where agents just have concurrent execution of behaviors. However, sometimes we need them to act as a whole or to be controlled by another agent, for example, following a commander of a platoon. This involves building agent hierarchies and concurrent representations that are supported by the CRAA.

8.3.2 Cross-Resolution Agent Architecture

Sometimes we need both low and high resolution agents to solve problems. As an extension of the mass agent architecture, the CRAA supports concurrent execution of agent behaviors. The CRAA also supports concurrent agent representations as shown in Figure 8.6, having perception and action implemented at different resolution levels for both the commander and soldiers. The agent plays not only the role of a commander (an LRA in the horizontal layered architecture), but also soldiers (HRAs in the horizontal layered architecture).

Now let us suppose a scenario in which a commander leads a platoon to accomplish the mission of clearing all mines distributed in a battlefield. As shown in Figure 8.7, assume initially only the LRA exists to model the platoon. The commander may order his soldiers to patrol together as a team (step 1 in Figure 8.8). If any mine is discovered (step 2 in Figure 8.8), a decision will be made by the commander and an agreement is achieved (step 3 in Figure 8.8). One (step 5 in Figure 8.8) or more soldiers (step 6

Figure 8.6: The CRAA agent for MSG

in Figure 8.8) will separate from the platoon to deal with the mine, and the remaining soldiers still continue to patrol together. This is the case where partial disaggregation will be carried out. In another case, once they discover mines, they all jump down from the truck and start to clear them, that is the case where full disaggregation is carried out (step 4 in Figure 8.8).

In this example, initially the soldiers do not have individual desires, and their deliberation processes do not exist. Once a mine is discovered, a disaggregation function is invoked to initialize the newly disaggregated soldiers that separate from the platoon. From then on, the separated soldiers are executed concurrently as sub-behaviors of the CRAA and they will have their own actions. However, the commander still keeps control over them, adjusting their behaviors if necessary through the CRAA. For example, if two soldiers detect two mines at the same time, they may attempt to pick up the same

Figure 8.7: The CRAA agent for MSG before disaggregation

Figure 8.8: Disaggregation procedure

mine. The commander can adjust one of the soldiers' behaviors to pick up the other mine instead of the same one, in order to increase their working efficiency. The CRAA agent for the MSG scenario after disaggregation is the same as that shown in Figure 8.6. Thus it is also attractive to use the CRAA to support aggregation/disaggregation, which is adaptive for fast changing battlefields.

8.3.3 Disaggregation Inconsistency Problem

As discussed in Section 8.3.2, dynamic resolution changes such as aggregation and disaggregation are required in some cases, especially for military simulations. However, inconsistency may arise during a sequence of agent resolution transitions.

Figure 8.9 gives a typical example where consistency may be violated during the HRA $\rightarrow$ LRA $\rightarrow$ HRA transition process. In this scenario, a two dimensional environment is initialized, with a team of soldier agents on the left side of a river. They can only cross the river using a bridge, which is some distance away. During this march, a sequence of changes in agent resolution may occur.

Figure 8.9: Disaggregation inconsistency

Originally, the team of soldiers are in disaggregated state in which every soldier is represented by a distinct HRA. After aggregation, an LRA is set to take charge of them

as a platoon instead of as individual soldiers. This results in the representations being converted from a high resolution level to a low resolution level. Looking at the scenario given in Figure 8.9, in the aggregated state, the platoon is still on the left side of the river, but it is hard to say whether each soldier is on the left or right side of the river. Thus if the soldiers disaggregate from the aggregated state into the disaggregated state of individual soldiers again, some of the soldiers may appear on the other side of the river. This is obviously incorrect as no bridge is here. The soldiers' consistency of position is violated during this sequence of dynamic changes in agent resolution. We should find some way to solve this problem.

Let us go back to the original condition of this scenario. To simplify the discussion, we assume aggregation is only allowed if all the soldiers in the platoon are on the same side of the river. Initially each of the soldiers should have a *position* attribute (x_i, y_i) ($i = 1, 2, \dots, n$) indicating his position. After the aggregation, the platoon will no longer have a *position* attribute for each individual soldier, but only one attribute $position^{agg} = (x^{agg}, y^{agg})$ indicating the whole platoon's position. The value of the platoon's position is the average value of all the soldiers' positions obtained using the following two mapping functions:

$$x^{agg} = \frac{1}{n} \sum_{i=1}^n x_i, \text{ and } y^{agg} = \frac{1}{n} \sum_{i=1}^n y_i$$

When the soldiers disaggregate again, the $position^{agg}$ may recover to (x'_i, y'_i) ($i = 1, 2, \dots, n$). Obviously we should ensure $x^{agg} = \frac{1}{n} \sum_{i=1}^n x'_i$, and $y^{agg} = \frac{1}{n} \sum_{i=1}^n y'_i$, thus the average value of their *position* will keep equal to the position of the platoon. However, these mapping functions are still insufficient to guarantee the correctness, as arbitrary values of (x'_i, y'_i) may be chosen. Some of them may be on the opposite side of the river. The violation of position consistency may easily occur, especially in military simulations where dynamic changes might happen very frequently.

We can follow the approach introduced in Section 7.4, as implemented in the CRAA. Initially each soldier, which is an HRA, is on the left side of the river. This is indicated by a *riverside* attribute, which depends on the soldier's *position* attribute. Since the soldier's *riverside* can be calculated by his position, we define a mapping function *side* from the *position* attribute to the *riverside* attribute, such that $riverside = side(position)$. The *side* function is the same for the platoon layer and soldier layer.

Figure 8.10 gives an ADG for these agent attributes, in which *dis* is used to indicate the disaggregation operation. For maintaining the consistency, the following formula should be ensured:

$$side \circ dis = dis \circ side$$

The disaggregation causes new values of (x'_i, y'_i) to be chosen as described above. The mapping function for the *riverside* attribute is *identity*. Hence, to maintain consistency, the following should be ensured:

$$side(x'_i, y'_i) = side(\frac{1}{n} \sum_{i=1}^n x'_i, \frac{1}{n} \sum_{i=1}^n y'_i)$$

So after a sequence of aggregation and disaggregation, all the agents should still be on the same side of the river.

Consistency enforcers are applied to enforce these mapping functions on the *position* and *riverside* attributes during the sequence of agent resolution transitions. The consistency enforcers are executed every time disaggregation occurs to ensure that after transition the values of (x'_i, y'_i) must satisfy the above equation. In this way, after restoring the values of *position* for all HRAs, the transition phase will not give inconsistencies about the soldiers' *riverside* attributes. In conclusion, mapping functions and consistency enforcers can be used to solve the disaggregation inconsistency problem.

Figure 8.10: Dependency graph for the disaggregation phase

8.4 Experimental Results

We carry out several experiments to evaluate the performance of the mass agent and CRAA agent. First we compare the execution time required for atomic agents, mass agents and CRAA agents. We have obtained the results using a set of PCs with 2.20GHz Pentium 4 processor and 1.00GB RAM. Each of the PCs has two processors, thus two HLA federates are placed on one machine.

Atomic	Mass	CRAA
ten soldiers as atomic agents	ten soldiers per mass agent	a commander and ten soldiers per CRAA agent

Table 8.2: Atomic agents, mass agents and CRAA agents

The three different cases of the configuration are presented in Table 8.2 and Figure 8.11. A mass agent is created representing ten soldiers of the MSG prototype system (case 2 in Figure 8.11), or a CRAA agent is set up representing a platoon including a commander and ten soldiers (case 3 in Figure 8.11). We compare these two cases with the original one where ten atomic agents represent ten soldiers respectively (case 1 in Figure 8.11). In all cases, the environment federate and the JADE's platform and main container are located on one separate machine. The agent federates will be executed with

one federate per CPU on the other machines and they are connected using the HLA/RTI via a LAN.

Figure 8.11: Federate configuration for comparing atomic agents, mass agents and CRAA agents

Figure 8.12 shows the results. The horizontal axis in this figure indicates the number of agent federates, and the vertical axis indicates the average value of the execution time spent for each simulation cycle. We tested up to eight federates, so that there are eighty soldiers at most executing at the same time. Obviously the mass agent architecture provides better efficiency than using atomic agents. At the same time, we can find that the CRAA does not cause much overload in efficiency compared with the mass agent. The results also show that the performance is scalable as the number of federates increases.

Agents' disaggregation times for atomic agents and mass/CRAA agents have also been

Figure 8.12: Execution time of atomic agents, mass agents and CRAA agents

evaluated on a single PC with two 2.20GHz Pentium 4 processors. Figure 8.13 shows the execution times required for the disaggregation from an atomic LRA to atomic HRAs. The horizontal axis indicates the number of atomic agents that disaggregate from the original agent. The vertical axis indicates the average execution time over a number of executions in milliseconds. Different numbers of HRAs, such as soldiers, can separate from the agent that represents an LRA such as a platoon. As we see, if only one or two soldiers disaggregate from the platoon, the time taken is very short. However, when the number of soldiers increases, the time required also increases. This is because creating more new atomic agents needs much more time for initializing their states and synchronizing their behaviors. The case that five soldiers disaggregate from the platoon is still acceptable. However, if ten or more agents disaggregate at the same time, the latency becomes huge. A conclusion can be made that during the simulation execution, it is better that only a few atomic agents disaggregate from the LRA, thus the disaggregation transition will not become a heavy burden on the required computing resources.

Figure 8.13: Disaggregation time for atomic agents

We can use mass agents instead of using atomic agents for saving computing resources and improving efficiency. Figure 8.14 shows the case where a mass agent is applied instead of initializing new atomic agents every time. When different numbers of soldiers need to separate from a platoon, creating new sub-behaviors inside the platoon to represent the soldiers is better than creating new soldier agents as atomic agents. Since the time spent on creating behaviors inside an existing agent is much less than the time spent on initializing and creating new atomic agents, obviously the mass agent architecture can provide a much more efficient implementation of the CRAA ideas than atomic agents.

In Section 7.4, we discussed the need for both concurrent agent representations and agent resolution transition from the simulation modeling perspective. From the point of view of execution efficiency, we may also need aggregation/disaggregation transition to save computing resources. For MAS simulations, we may not need HRAs all the time. In some cases, it may be better to create them when necessary, and delete them if they are not required any more. Although the aggregation/disaggregation transition takes some

Figure 8.14: Disaggregation time for a mass agent

time, it is still possible to achieve better efficiency. The results in Figure 8.15 demonstrate this, and it is interesting for a developer to decide when to use aggregation/disaggregation and when to use concurrent agent representations.

Figure 8.15 shows the execution time of an implementation that represents a platoon with both high and low resolution levels throughout, compared with the case where the aggregation/disaggregation transition is applied. The experiment is carried out on a single PC with two 2.20GHz Pentium 4 processor and 1.00GB RAM. We have tested the case where twenty soldiers are used as the HRAs and one commander as the LRA. They jointly execute together for one hundred simulation cycles, during which time it is possible they carry out aggregation and disaggregation resolution transitions. The horizontal axis in this figure indicates the percentage of the time HRAs are required compared to the whole simulation time, and the vertical axis indicates the total execution time in milliseconds. If the implementation always uses concurrent agent representations from the beginning to the end, the execution time keeps a constant value. If we allow agents to disaggregate and

Figure 8.15: Execution time of alternative solutions

aggregate when necessary, the value of the execution time will increase along the direction of the horizontal axis, namely, using aggregation/disaggregation helps to save computing resources when the percentage of the time HRAs are required is under approximately 70% of the whole simulation time.

In conclusion, if the percentage of the time HRAs are required is low during the execution, agent resolution transition can bring better execution efficiency than that of supporting concurrent representations throughout. Of course the point of balance between concurrent representations and resolution transition may vary according to the number of HRAs and the computing resources required in practice. We recommend developers to use concurrent agent representations to avoid frequent occurrences of agent resolution transition.

8.5 Summary

The idea of mass agent has been introduced to support the concurrent execution of behaviors of peer agents that share the same set of decision-making rules. We have presented the cross-resolution agent architecture (CRAA) to extend the mass agent architecture. The CRAA implements the idea of concurrent agent representations as illustrated in the previous chapter, and involves building a hierarchy of agents across two or more agent layers. Consistency enforcers have been used to maintain consistency between representations at different resolution levels. The CRAA also supports agent resolution transition and solves the problem of aggregation/disaggregation inconsistency. Consistency enforcers have also been applied here to ensure the consistency during a sequence of agent resolution transitions. In conclusion, the CRAA significantly increases the flexibility of our MAS simulations.

Chapter 9

Conclusions

9.1 Achievements and Contributions

In this project, we develop a generic framework for autonomous agents in distributed simulations, which distinguishes our work from those who apply parallel and distributed simulation techniques to the simulation of agent architectures or those who use agents as a way of controlling simulations. To date, most of research in the cross field of agents and parallel/distributed simulation has focused on agent simulation, namely, to use simulation to understand an agent behavior and to investigate the impact of alternative architectures.

There exist many benefits of combining agent technology and parallel and distributed simulation. Firstly, when using agents as entities in a simulation or virtual environment, rapid and accurate decision-making can be achieved. Secondly, the technology of distributed simulation increases the interoperability and reusability of agents. The integration of agents and distributed simulation benefits not only the research area of distributed simulation but also that of agents.

In this thesis, we present an approach in which autonomous agents are integrated into distributed simulations. The novelty of our research comes from the way of integrating autonomous agents into DVEs. When utilizing agents in DVEs, many issues are involved

including agent architectures, integration approaches, agent communication, causality and consistency, etc. We address these issues from the perspective of agents, and also from the point of view of distributed simulations.

A general architecture is presented, with both the high level agent-specific services and the low level infrastructure conforming to recognized standards. In this architecture, a FIPA-compliant agent toolkit — JADE is selected to support agents and their communication language, and the HLA is utilized as an interoperability framework. This provides the possibility that agents and participants located in different geographical locations can share a common virtual world. At the same time, a special kind of federate — the gateway federate is constructed to achieve seamless connection of the JADE platform and the HLA. Two interfaces are developed between the gateway federate and its agent/agents to support a general agent architecture in which agents can receive inputs from the environment through sensors and acts on the environment through effectors. With these facilities, a complete interaction mechanism for situated agents comes into being, and a sense-deliberate-react agent architecture is obtained.

By using the gateway federate, HLA services can be mapped to the requirements of a distributed multi-agent simulation, including time request and advance, attributes publication and subscription, object registration and discovery, and interaction sending and receiving etc. Every agent is attached to a gateway federate, however it is also possible to have a group of agents attached to the same gateway federate. Different gateway federates can be placed in either the same physical machine or different machines, which facilitates the flexibility and extensibility of this integration architecture. Our proposal meets the requirement that an agent simulation component such as a developed gateway federate can be reused in other simulations or embedded in other environments.

We implement a prototype system called MSG for testing the feasibility of our approach for integrating agents into a distributed simulation. In the MSG, several soldiers

perform the tasks of detecting mines in an environment and clearing them. The soldiers and the environment are physically located on different computers distributed over a network and connected by the HLA Runtime Infrastructure. The soldiers are enabled to handle situations where only partial information is available. This prototype system provides a basis for further investigations of research issues and a way of evaluating performance.

In this project, the issue of agent communication is also addressed, as one of the key characteristics of MASs. Agents are enabled to communicate to share information and achieve a certain level of coordination in distributed simulations. The benefits of agent coordination in distributed simulations are demonstrated by experimental results. In the scenario of MSG, compared to the situation where no ACL messages and coordination are applied, communication among soldier agents is demonstrated to provide noticeable gains in the agents' working efficiency, especially when the target mines distributed in the environment are dense and clustered.

When utilizing agents as entities in distributed simulations, it is essential to investigate appropriate strategies for synchronization, especially for multi-agent systems where multiple agents perform tasks in a shared virtual environment. Since the agent's deliberation phase is a time and space consuming process, its execution time cannot be predicted. Latch objects are developed in the prototype system, in order to preserve causality and maintain consistency among the soldiers and their environment.

It is also a problem to guarantee the consistency of a distributed simulation in which agents communicate with each other. One of the HLA rules states that, during a federation execution, all exchange of FOM data among federates shall occur via the RTI. Thus it is not acceptable to simply use the existing message transmission mechanism which is provided by some agent platforms. We introduce mailbox agents to solve this problem. Two constraints are made on agent-to-agent communication in distributed simulations

to ensure consistency, and corresponding behaviors of mailbox agents are designed. Although the gateway federate, autonomous agents and mailbox agent are running on the same physical machine, they still execute as different threads, and it is therefore necessary to synchronize them correctly. A strategy is developed to meet this requirement, letting them execute concurrently and interoperate with each other. The mechanisms for agent communication including the mailbox agent and the synchronization strategy are demonstrated to be effective by experimental results.

When various kinds of agents are involved in a distributed simulation, the relationships among agents are complex. Agents may have different representations for the same phenomenon at different resolution levels. Thus the issue of multi-resolution modeling for MAS simulations is also investigated. Three kinds of hierarchies are explored for MAS simulations according to the heterogeneous control among them. We have extended the concept of concurrent representations proposed by Natrajan [43] in the specific area of agent simulation, and have presented the idea of concurrent agent representations to enable jointly-executing agents to have representations at different resolution levels. The consistency of representations is discussed for both concurrent agent representations and agent resolution transition. The consistency enforcers introduced by Natrajan [43] are also adapted here to maintain agent representation consistency during the simulation execution, invoking application-specific mapping functions that translate changes in one agent attribute to changes in another.

The mass agent architecture is introduced to wrap peer agents that share the same set of decision-making rules within a single agent architecture. We also extend the mass agent to a novel MAS architecture — CRAA which is a cross-resolution agent architecture that implements the idea of concurrent agent representations. The CRAA involves building a hierarchy of agents across two or more agent layers. This is valuable especially for military simulations in which an army corps may break down into more and more de-

tailed subdivisions hierarchically. In the CRAA, agents with representations at different resolution levels are linked together. This architecture can also support dynamic changes of agent resolution including aggregation and disaggregation. In both cases, consistency enforcers are applied to ensure there is no violation of representation consistency. The CRAA algorithms are investigated for two or more agent layers.

In summary, the contributions of our work are the following:

- Integration approach:
 - A study of the cross field of combining agents with parallel and distributed simulation.
 - A general architecture for integrating autonomous agents into a distributed virtual environment.
 - A gateway federate for seamless connection of the JADE platform and the HLA/RTI.
 - Methods for synchronizing agent/agents' execution with the gateway federate.
 - A prototype system MSG as a test bed.
- Agent communication in distributed simulations:
 - Specification of assumptions and constraints on agent communication to ensure consistency.
 - A mailbox agent and associated algorithms for message transmission among agents in distributed simulations.
 - Methods for synchronizing the mailbox agent with other agents and the gateway federate.
 - An investigation on agent coordination in the MSG scenario.

- Multi-resolution modeling:
 - A taxonomy for hierarchies of MASs.
 - The introduction of concurrent representations in the specific area of agent simulation.
 - Adapted techniques for maintaining the consistency of concurrent agent representations based on the ADG and mapping functions.
- A cross-resolution agent architecture:
 - The mass agent for concurrent execution of peer agents that share the same set of decision-making rules.
 - The CRAA for supporting concurrent agent representations and dynamic changes in agent resolution.
 - Algorithms for the cross-resolution agent architecture for two and more agent layers.

9.2 Future Work

There are still some interesting issues that need to be investigated in the future. First, all the current work is based on time-stepped simulation. It is valuable to extend the work to event-driven simulation that uses a different mechanism to advance simulation time. In event-driven simulation, variables are only updated when an event occurs. This leads to more efficient simulation executions in situations where the occurrence of events is irregular as compared to time-stepped simulation where new values for variables are computed at each time-step. It is also interesting to investigate how the simulation scales if more and more agents are added into the MSG.

Secondly, for a real-time system where human participants are embedded into the virtual environment, instead of allowing an agent to deliberate for as long as it wishes, time constraints may be impressed upon the agent. Sometimes agents must make decisions under strict time pressures. Thus when developing agents as entities in a distributed virtual environment, appropriate strategies will be needed for addressing this issue. It will be necessary to explore agent behaviors that can deal with problem solving under real-time constraints.

Thirdly, we only address issues involving stationary agents in this thesis. However, mobile agents are also very commonly used in practice. For distributed simulations in which multiple agents are developed to represent entities, the migration of agents will cause more complicated problems concerning causality and consistency. Consider a situation in which an agent sends a message to assign a task to another agent. The latter may accept the assigned task and acknowledge the former about the acceptance of this task. However, the former agent may migrate to another geographically different location during this time, thus the mailbox agent may not find the correct receiving address for the acknowledgement message. Thus additional algorithms will be required to ensure the consistency before and after the migration.

Finally, the CRAA enables agents with representations at different resolution levels to be linked and executed together, but this only covers the situation where a group of hierarchical agents are in the same agent federate. If the cross-agent relationships of concurrent agent representations involves agents that are from different agent federates, the CRAA will need to be extended to support agent interactions between different federates to enforce the consistency and to make behavior adjustments. It will be beneficial if an architecture and associated algorithms which are more flexible and extensible could be developed to be applied in this case.

9.3 Summary

In this thesis, a general architecture has been presented for integrating agents and distributed simulations. The gateway federate has been developed to increase the interoperability and reusability of MAS simulations, and the mailbox agent has been developed to enable agents to communicate with each other in distributed simulations. By the implementation of the prototype system, communication has been demonstrated to be effective for achieving better working efficiency for the agents in distributed simulations. The issue of multi-resolution modeling has also been investigated, including both concurrent representations of agents and agent resolution transition. An advanced agent architecture — CRAA has been developed, linking agents with representations at different resolution levels. This greatly enhances the flexibility of our architecture for MASs in distributed simulations.

Bibliography

- [1] Agent Theories, Architectures, and Languages (ATAL), The International Workshop Series. <http://www.atal.org/>.
- [2] OMG MASIF Standard, OMG Agent Platform Special Interest Group, Mar 1998. <http://www.objs.com/agent/>.
- [3] FIPA Agent Management Specification. Technical Report SC00023J, Foundation for Intelligent Physical Agents, Dec 2002. <http://www.fipa.org/>.
- [4] IEEE 1278. Standard for Information Technology, Protocols for Distributed Interactive Simulation (DIS), 1993.
- [5] IEEE 1516. Standard for Modeling and Simulation (M&S), High-Level Architecture, 2001.
- [6] J. Andersson and S. Löf. HLA as conceptual basis for a multi-agent environment. Technical Report 8TH-CGF-033, Pitch Kunskapsutveckling AB, 1999.
- [7] F. Bellifemine, A. Poggi, and G. Rimassa. JADE – a FIPA-compliant agent framework. In *Proceedings of the Practical Applications of Intelligent Agents and Multi-Agents*, pages 97–108, London, UK, 1999.
- [8] M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, (4):349–355, 1988.

-
- [9] BT. The ZEUS agent building toolkit. <http://www.labs.bt.com/projects/agents/zeus/>.
- [10] B. Burmeister and K. Sundermeyer. Cooperative problem-solving guided by intentions and perception (abstract). *SIGOIS Bull.*, 13(3):10, 1992.
- [11] W. T. Cai, P. Xavier, S. J. Turner, and B. S. Lee. A scalable architecture for supporting interactive games on the Internet. *16th IEEE/ACM/SCS Workshop on Parallel and Distributed Simulation (PADS 2002)*, pages 60–67, 2002.
- [12] F. Chantemargue and B. Hirsbrunner. A collective robotics application based on emergence and self-organization. Technical report, PAI group, Fribourg, Switzerland, 1999.
- [13] P. Ciancarini and M. Wooldridge, editors. *Agent-Oriented Software Engineering: The State of the Art*, volume 1957 of *Springer-Verlag*. LNAI, 2001.
- [14] T. Dagaëff, F. Chantemargue, and B. Hirsbrunner. Emergence-based cooperation in a multi-agent system. AXE project, grants 21-43558.95 and 21-47262.96, Swiss National Funds for Scientific Research, 1997.
- [15] P. K. Davis and R. Hillestad. Families of models that cross levels of resolution: issues for design, calibration and management. In *WSC '93: Proceedings of the 25th Winter Simulation Conference*, pages 1003–1012. ACM Press, 1993.
- [16] T. Eiter and V. Mascardi. Comparing environments for developing software agents. INFSYS Research Report 1843-01-02, Institut und Ludwig Wittgenstein Labor für Informationssysteme, Austria, March 2001.
- [17] T. Finin and R. Fritzson. KQML: a language and protocol for knowledge and information exchange. *Proceedings of 19th International Distributed Artificial Intelligence Workshop*, pages 127–136, 1994.
-

-
- [18] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an agent communication language. *Proc. 3rd Int. Conf. On Information and Knowledge Management (CIKM '94)*, ACM Press, pages 456–463, November 1994.
- [19] E. J. Friedman-Hill. Jess, the expert system shell for the Java platform. Technical Report SAND98-8206, Sandia National Laboratories, 2003.
- [20] R. M. Fujimoto. Time management in the High Level Architecture. *Simulation*, 71(6):388–400, 1998.
- [21] R. M. Fujimoto. *Parallel and Distributed Simulation Systems*. John Wiley & Sons, Inc., Wiley, 2000.
- [22] L. Gautier and C. Diot. Design and evaluation of MiMaze, a multi-player game on the Internet. In *ICMCS '98: Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 233–236, Washington, DC, USA, 1998. IEEE Computer Society.
- [23] M. Georgeff, B. Pell, M. Pollack, M. Tambe, and M. Wooldridge. The belief-desire-intention model of agency. In *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 1–10. Springer-Verlag: Heidelberg, Germany, 1999.
- [24] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, 1987.
- [25] N. Guarino. Formal ontology, conceptual analysis and knowledge representation. In *Special issue of the International Journal of Human and Computer Studies*, volume 43, pages 625–640, 1995.
-

-
- [26] V. Gyurjyan, D. Abbott, G. Heyes, E. Jastrzembski, C. Timmer, and E. Wolin. FIPA agent based network distributed control system. In *ECONF*, volume C0303241, page THGT009, La Jolla, California, 2003.
- [27] M. N. Huhns and L. M. Stephens. Multiagent systems and societies of agents. In *Multiagent systems: a modern approach to distributed artificial intelligence*, pages 79–120, Cambridge, MA, USA, 1999. MIT Press.
- [28] N. R. Jennings. Specification and implementation of a belief, desire, joint-intention architecture for collaborative problem solving. *Journal of Intelligent and Cooperative Information Systems*, 2(3):289–318, 1993.
- [29] N. R. Jennings. Agent-oriented software engineering. In *Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World : Multi-Agent System Engineering (MAAMAW-99)*, volume 1647, pages 1–7. Springer-Verlag: Heidelberg, Germany, 1999.
- [30] N. R. Jennings, Katia Sycara, and M. Wooldridge. A roadmap of agent research and development. In *Autonomous Agents and Multi-Agent Systems*, volume 1, pages 7–38, Kluwer Academic Publishers, Boston, 1998.
- [31] N. R. Jennings and M. J. Wooldridge, editors. *Agent technology: foundations, applications, and markets*. Springer. UNICOM, Berlin, Heidelberg and New York, 1998.
- [32] N. R. Jennings and M. R. Wooldridge, editors. *Agent Technology Foundations, Applications and Markets*. Spring-Verlag. 1998.
- [33] Nicholas R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 177(2):277–296, 2000.

-
- [34] A. W. F. Lee, W. Sweldens, P. Schröder, L. Cowsar, and D. Dobkin. MAPS: Multiresolution adaptive parameterization of surfaces. *Computer Graphics Proceedings (SIGGRAPH 98)*, pages 95–104, 1998.
- [35] M. Lees, B. Logan, and G. Theodoropoulos. Agent, computer games and HLA. In *European Simulation Interoperability Workshop (EuroSIW)*, number 02E-SIW-051, UK, 2002.
- [36] M. Lees, B. Logan, G. Theodoropoulos, and T. Oguara. Simulating agent-based systems with HLA: the case of *SIM_AGENT* – part II. In *European Simulation Interoperability Workshop (Euro-SIW)*, number 03E-SIW-076, UK, 2003.
- [37] B. Logan and A. Sloman. Agent route planning in complex terrains. Technical Report CSRP-97-30, School of Computer Science, University of Birmingham, 1997.
- [38] B. Logan and G. Theodoropoulos. The distributed simulation of multi-agent systems. In *Proceedings of IEEE*, volume 89, pages 174–186, UK, 2001.
- [39] V. Mascardi. An agent-based approach to distributed simulation, distributed prototyping and software integration. Technical report, University of Genova, Italy, 1999.
- [40] Rob Minson and Georgios Theodoropoulos. Distributing RePast Agent-Based Simulations with HLA. In *European Simulation Interoperability Workshop (EuroSIW)*, number 04E-SIW-046, Edinburgh, June 2004.
- [41] K. L. Morse. Interest management in large scale distributed simulations. Technical Report 96-27, Department of Information and Computer Science, University of California, Irvine, 1996.
- [42] J. P. Müller, M. Pischel, and M. Thiel. Modelling reactive behaviour in vertically layered agent architectures. In *Proceedings of the Workshop on Agent Theories*,

- Architectures, and Languages on Intelligent Agents*, pages 261–276. Springer-Verlag New York, Inc., 1995.
- [43] A. Natrajan. *Consistency Maintenance in Concurrent Representations*. PhD thesis, University of Virginia, January 2000.
- [44] C. J. Petrie. Agent-based software engineering. In *Agent-Oriented Software Engineering, Lecture Notes in AI 1957*, pages 58–76. Springer-Verlag, 2001.
- [45] M. Pollack and M. Ringuette. Introducing the Tileworld: Experimentally Evaluating Agent Architectures. In Thomas Dietterich and William Swartout, editors, *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 183–189, Menlo Park, CA, 1990. AAAI Press.
- [46] S. J. Poslad, R. A. Bourne, A. L. G. Hayzelden, and P. Buckle. Agent technology for communications infrastructure: an introduction. In Alex L. G. Hayzelden and Rachel A. Bourne, editors, *Agent Technology for Communication Infrastructures*, pages 1–18. John Wiley & Sons Ltd., 2001.
- [47] E. Puppo and R. Scopigno. Simplification, LOD, and multiresolution - principles and applications. In *EUROGRAPHICS'97 Tutorial Notes (ISSN 1017-4656)*, volume 16, Switzerland, 1997. Eurographics Association.
- [48] R. Radhakrishnan and Philip A. Wilsey. Ruminations on the implications of multi-resolution modeling on DIS/HLA. In *3rd International Workshop on Distributed Interactive Simulation and Real-Time Applications (DIS-RT)*, pages 101–108, Maryland, 1999.
- [49] A. S. Rao and M. P. Georgeff. BDI agents: from theory to practice. Technical report, Cooperative Research Centre for Intelligent Decision Systems under the Australian Government's Cooperative Research Centres Program, 1995.

-
- [50] P. F. Reynolds, A. Natrajan, and S. Srinivasan. Consistency Maintenance in Multiresolution Simulations. *ACM Transactions on Modeling and Computer Simulation*, 7(3):368–392, July 1997.
 - [51] S. J. Russell and P. Norvig, editors. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003.
 - [52] J. R. Searle. Speech acts: an essay in the philosophy of language. Technical report, Cambridge University, UK, 1969.
 - [53] X. J. Shen, R. Hage, and N. Georganas. Agent-aided collaborative virtual environments over HLA/RTI. Technical report, Multimedia Communication Research Lab (MCRLab), School of Information Technology and Engineering, University of Ottawa, Canada.
 - [54] A. Sloman and R. Poli. SIM AGENT: A toolkit for exploring agent designs. *Lecture Notes in Computer Science*, 1037:392–406, 1996.
 - [55] I. Soto, M. Garijo, C. A. Iglesias, and M. Ramos. An agent architecture to fulfill real-time requirements. *Agent 2000*, pages 475–482, 2000.
 - [56] G. Tan and L. Xu. An agent-based DDM filtering mechanism for HLA. *Special Issue on Software Agents and Simulation, Simulation*, 76(6):329–344, June 2001.
 - [57] A. M. Uhrmacher and K. Gugler. Distributed, parallel simulation of multiple, deliberative agents. In *Proceedings of the 14th Parallel and Distributed Simulation Conferenc (PADS’2000)*, pages 101–110, Bologna, May 2000.
 - [58] A. M. Uhrmacher and K. Gugler. A component-based simulation layer for JAMES. In *18th Workshop on Parallel and Distributed Simulation*, pages 115–122, Kufstein, Austria, May 2004.
-

-
- [59] G. J. Valentino and K. L. Morse. An agents toolkit to support distributed simulations. Technical Report SBIR Contract M67004-97-C-0038, Orlando, FL, 1997.
- [60] F. Wang, S. J. Turner, and L. H. Wang. Integrating agents into HLA-based distributed virtual environments. In *4th Workshop on Agent-Based Simulation (ABS2003)*, pages 9–14, Montpellier, France, April 28-30 2003.
- [61] L. H. Wang, S. J. Turner, and F. Wang. Interest management in agent-based distributed simulations. In *7th IEEE International Workshop on Distributed Simulation and Real-Time Application (DS-RT 2003)*, pages 20–27, Delft, The Netherlands, October 2003.
- [62] G. S. Wasson and W. Martin. Multi-tiered representation for autonomous robots. In *Proceedings of SPIE, Mobile Robots XIII and Intelligent Transportation Systems*, volume Volume 3523, pages 4–12, 1998.
- [63] M. Wooldridge. Agent-based software engineering. *IEE Transactions on Software Engineering*, 144(1):26–37, February 1997.
- [64] M. Wooldridge and N. R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.
- [65] Y. Y. Zhang, L. Guo, and N. D. Genorganas. AGILE: an architecture for agent-based collaborative and interactive virtual environments. Technical report, Multimedia Communications Research Laboratory, University of Ottawa, Ottawa, Ontario, Canada, 1999.
- [66] S. P. Zhou, W. T. Cai, B. S. Lee, and S. J. Turner. Time-space consistency in large scale distributed virtual environments. *ACM Transactions on Modeling and Computer Simulation*, 14(1):31–47, 2004.
-