Facilitating Cooperation in Multi-agent Robotic Systems

Toktam Ebadi

Supervisors

Dr. Maryam Purvis and Professor Martin K. Purvis

A thesis submitted for the degree of

Doctor of Philosophy

At the university of Otago, Dunedin,

New Zealand

March 7, 2012
This thesis presents a new framework for developing, coordinating, and managing a system of situated agents for operation in distributed spatial environments. The framework was developed using agent-oriented software engineering techniques that consider concurrent features of real systems. It provides a realistic development environment that ensures reliable deployment of coordination schemes and mechanisms in real contexts.

The framework uses multi-threaded agents so all the agents are executed concurrently, and each has control over its own behaviour. Such a model allows us to capture the behaviour of agents under realistic conditions where all the agents act concurrently. In addition, the framework employs Coloured Petri Net techniques to model multiple, concurrent conversations in which a given agent may be engaged.

In this framework agents must resolve complex and interdependent tasks. A Coloured Petri Net modelling technique was used to specify task plans and for the support of mechanisms both for the agents to identify task requirements and for managing the order in which various subtasks are performed. Our framework also enables the developed agents to identify subtasks that must be executed concurrently.

The framework takes into account the unreliability of agent communication in real, physical environments by employing a timeout mechanism to test situations when communication responses are lost or delayed. It uses an environmental system time to accommodate the time delay associated with agents’ responses in a distributed multi-agent system.

Any agent framework must support the ability of agents to adapt according to the new circumstances. Our framework enables the agents to adapt in various
ways. Agents can interact for solving various problems by changing their roles. Agents can change their roles at run-time. In addition, certain strategies were designed for agents to show how they can change their cooperation strategy in different situations to maximise their reward. A reinforcement learning mechanism was used to enable the agents to adopt the best strategies under changing circumstances.

The framework that we have developed uses the Opal agent platform which complies with standards provided by Foundation for Intelligent Physical Agent (FIPA). Agents in the framework use asynchronous communication, and agents’ messages are sent over a network. In addition, all the messages in the system use FIPA agent communication languages.

By providing a design environment that enables the direct deployment of developed methods on real, instantiated agent robot systems, this work enhances the robustness of the agent-oriented software engineering process for mobile robots. A cooperative task execution scenario was tested using commercially available mobile robots (Garcia). The experiment demonstrated the efficiency and effectiveness of our framework in developing and deploying robots organisational models in physically distributed scenarios.
Acknowledgements

This thesis would have not been possible without the support and encouragement of many people whom I would like to sincerely thank.

First and foremost, thank you to my supervisors, Dr. Maryam Purvis and Professor Martin Purvis for providing invaluable support and advice throughout the course of this thesis. Your support and guidance over the last five years that I have known you has been unbelievable. For that and everything else you have done, I am eternally grateful. Thank you!

I also would like to thank Mariusz Nowostawski, Mark George, Vincent Goh and Christopher Frantz for kindly providing help and professional advice at the time of need. I also would like to thank Melanie Middlemiss, Noria Foukia, Stephen Cranefield, Michael Winikoff, Tony Bastin Savarimuthu and Tony Moore for their expert advice on various parts of this thesis, and some of my publications.

Thanks to everyone within the department of Information Science especially guys in TSG, including Graeme Roxburgh, Graham Copson, Brendon Sly and Peter George for providing me with all the equipment required for this thesis in addition to their low-level technical support. A big thank you also goes to Stephen Hall-Jones, Heather Harris, Gail Mercer and Rachel Baker.

Thanks to my wonderful friends who have been very understanding as I have taken this journey. In particular, Anne and Dick Crush for having an open door with sympathetic ears. Your support has been incredible. I also would like to thank Chris Edwards and Beryl Lee for their proofreading skills that made this thesis a lot easier to read.

Thanks to my parents, Fatemeh and Hossein, and sisters, Sara, Samaneh, Sahar and Sanaz. You have been a constant source of support over these years. Despite
the jokes and digs you were right behind me encouraging me to succeed. I would like to thank Sara separately for her great help in the last few months. You mothered my children so I could focus on my thesis. Huge thank you!

Last, but by no means least, my heartfelt thanks to my husband Nasser for his unwavering love and support. I also would like to thank my two gorgeous girls, Yassamin and Hannah, for their wonderful patience and understanding.

Completing this thesis would not have been possible without the generous support of a University of Otago Postgraduate Scholarship (2006-2009).
List of Acronyms

**FIPA** Foundation of Intelligent Physical Agents

**MAS** Multi-agent System

**IP** Interaction Protocol

**UML** Unified Modelling Language

**AUML** Agent Unified Modeling Language

**FSM** Finite State Machine

**PN** Petri Net

**CPN** Coloured Petri Net

**ACL** Agent Communication Language

**KQML** Knowledge Query Manipulation Language

**MDT** Multi-criteria Decision making

**BDI** Belief-Desire-Intention

**VCP** Vacancy Chain Process

**RFID** Radio Frequency Identification

**AOSE** Agent Oriented Software Engineering

**UAV** Unmanned Aerial Vehicle

**PDA** Personal Digital Assistant

**ACM** Agent Construction Model

**OO** Object Oriented
Contents

Acronyms v

1 Introduction 1
  1.1 Motivation .......................................................... 1
  1.2 Research questions and original contributions ................. 3
    1.2.1 Different levels of agencies ............................... 4
    1.2.2 Modelling agents’ concurrent actions .................... 5
    1.2.3 Modelling tasks ............................................. 6
    1.2.4 Adapting in dynamic situations .......................... 6
  1.3 Publications ..................................................... 6
  1.4 Thesis structure ................................................ 8

2 Background 9
  2.1 Agent definition .................................................. 10
    2.1.1 Comparing agents and objects ............................. 11
    2.1.2 Agent properties .......................................... 12
  2.2 Multi Agent Systems ............................................ 15
    2.2.1 Why use multi-agent systems? ........................... 15
  2.3 Agent oriented software engineering ............................ 16
    2.3.1 Concurrency in multi-agent systems .................... 18
  2.4 Foundation for Intelligent Physical Agents .................... 19
  2.5 Agent Communication Languages ................................ 20
  2.6 Interaction protocols .......................................... 21
    2.6.1 FIPA protocols ........................................... 22
### 4.2.3 Task definition

- 

### 4.2.4 Agents attitude

- 

### 4.2.5 Reward mechanism

- 

### 4.3 Agent strategies

- **Nearest available** strategy
- **Best available** strategy
- **Best Possible** teammate strategy

### 4.4 Adaptation

- 

### 4.5 Experiments

- The effect of task time on agents reward where the agents employ various teammate selection strategies
- The effect of learning and adaptation on agent reward

### 4.6 Multi-threaded agent framework

- Concurrent issues in multi-threaded framework
- Multi-threaded framework description
- **Impatient** strategy

### 4.7 Multi-threaded framework experiments

- The effect of task time on agent reward
- The effect of referral on performance of agents with **Best possible** strategy
- The effect of learning on agents reward

### 4.8 Summary

- 

### 5 Modelling Agent’s Conversation

#### 5.1 Modelling agent

- Coloured Petri Nets

#### 5.2 Otago agent platform

- Opal conversation manager
- JFern

#### 5.3 Modelling agents’ roles

- Agents’ roles
5.3.2 Timeout .................................................. 102
5.3.3 Initiator role ........................................... 103
5.3.4 Helper role ............................................. 104
5.4 Task .......................................................... 106
5.4.1 Task selection strategies .............................. 106
5.5 Agent reward .............................................. 107
5.6 Experiments .................................................. 108
  5.6.1 The effect of teammate selection strategies on agents reward when task density is low .......................... 110
  5.6.2 The effect of teammate selection strategies on agents reward when task density is high .......................... 112
  5.6.3 The effect of agent learning on agents reward ........................................ 113
  5.6.4 The effect of agent task selection strategy on agents reward ........................................ 115
5.7 Summary .................................................... 116

6 Modelling Tasks ............................................. 118
  6.1 Task specification ......................................... 120
  6.2 Task CPN .................................................. 120
  6.3 Agent models ............................................. 123
    6.3.1 Initiator role ......................................... 123
    6.3.2 Task manager role .................................. 124
    6.3.3 Adaptive task manager role ......................... 126
    6.3.4 Helper role .......................................... 127
  6.4 Adaptation ................................................ 130
    6.4.1 Adopting the best strategy ......................... 130
    6.4.2 Adapting to unreliable situations .................. 131
  6.5 Experiments .............................................. 132
    6.5.1 Simulation parameters ............................... 134
    6.5.2 Adopting the best strategy ......................... 135
    6.5.3 The effect of agents learning about other agents’ reliability ......................... 137
  6.6 Summary .................................................. 139
List of Figures

2.1 An overview of FIPA specifications .............................................. 20
2.2 FIPA Request Interaction Protocol .............................................. 23
2.3 An example of a FSM formalism .................................................. 28
2.4 An example of modelling a conversation using Dooley graph .............. 30
2.5 FIPA Agent Management Reference Model .................................. 33

3.1 An example of a task tree for monitoring two regions ...................... 48
3.2 Task tree for concurrent scanning task .......................................... 50
3.3 The constraint network defined by a task tree ................................. 51
3.4 An example of a task precedence order graph ................................ 52

4.1 Example of various agent visibility ranges .................................... 76
4.2 Comparing the reward of various agent strategies under various task time
    constraints ................................................................. 78
4.3 Comparing the number of tasks that are fully completed for different strate-
    gies under various task time constraints ................................ 79
4.4 The impact of learning on agent reward ....................................... 80
4.5 The effect of agents’ strategies on agents reward under various task time
    constraints in the multi-threaded framework ............................. 86
4.6 Effect of referral on agents’ task completion performance ................ 88
4.7 The Effect of learning on agents reward using the multi-threaded framework 90

5.1 An example of a CPN model ...................................................... 95
5.2 Opal architecture ................................................................. 99
5.3 JFern snapshot ................................................................. 100
List of Tables

4.1 Various agent strategies, based on attitudes. An agent strategy is determined based on its attitudes. ................................................................. 68
4.2 System parameters ........................................................................... 77
4.3 Agents strategies in the multi-threaded framework. An agent strategy is determined based on its attitudes. ............................................. 85
4.4 Simulation parameters for the multi-threaded framework. All time values are given in milliseconds. ................................................................. 86

5.1 Simulation parameters. All time values are given in seconds. ............. 109
6.1 Simulation parameters. All time values are given in seconds. .............. 134
6.2 The population percentage for each strategy at point P1 and P2. ........... 136
List of Algorithms

4.1 Pseudocode for calculating an agent’s distance from a task. . . . . . . . . . . . 69
4.2 Pseudocode for Nearest available strategy. . . . . . . . . . . . . . . . . . . . 71
4.3 Pseudocode for Best available strategy. . . . . . . . . . . . . . . . . . . . . 73
4.4 Pseudocode for multi-threaded agents . . . . . . . . . . . . . . . . . . . . . 83
Chapter 1

Introduction

Cooperation involves individuals or groups that work together in order to achieve their individual or collective goals. Cooperation is essential in situations where an individual cannot perform a task by itself. Even in situations where individuals are capable of performing tasks, cooperation improves the performance of the overall system by allowing for parallel execution of multiple tasks. A joint, student project is a simple example of a cooperative activity where two or more individuals work together on a single project. More complex examples involve groups of software developers that all work together to design and implement a complex software application. Such complex scenarios require coordination among individuals.

Multi-agent software technology is inspired by human societies and biological systems, and investigates mechanisms and methodologies that can model and facilitate cooperation among agents. It is especially suitable for coordinated problem solving, since it can facilitate automated negotiations between distributed entities, and can also support more efficient management of distributed systems.

1.1 Motivation

Typically, individual agents are constrained to operate within environments bounded by limitations in computational resources and time, and are deployed in open environments. Agents may have different expertise (or capabilities) and act as self-interested entities to achieve their
corresponding goals (self-interested agents aim to fulfil their own goals). In such systems, it is not possible to specify a priori the contexts in which an agent might need to interact with another for its service requirements. Agents may have limited resources and may confront complex tasks that cannot be performed by a single agent. Therefore they must be able to interact with other agents in order to build a team that can solve such complex tasks. Hence, there is a need for effective coordination mechanisms that coordinate agents activities. Such coordination mechanisms must consider various constraints with respect to agents and tasks. Agent constraints may refer to the capabilities of the agents, while task constraints may include time constraints (i.e., the time given to agents to perform the tasks).

With respect to the domain of application of multi-agent systems, we note that such systems are suited to operate in situations that are complex and unpredictable. For example, social unrest and climate change are accelerating the occurrence of disasters requiring rapid response in sometimes dangerous or inaccessible conditions. At the time of writing this thesis, social unrest was evident in Egypt and Libya. In addition, two major natural disasters occurred in New Zealand (an earthquake in Christchurch that left 180 people dead) and Japan (earthquake and tsunami near the city of Sendai). In both situations a system that involved robotic agents that could autonomously cooperate and coordinate their activities would have been extremely beneficial. Specifically, in cases like the one in Japan, where there may be a chance of radiation leaks from a damaged nuclear reactor, the use of robotic agents for monitoring and possibly fixing the problem, could prevent humans operating in hazardous situations.

Several existing frameworks have been used for facilitating cooperation and coordination in multi-agent systems. Most of those frameworks however, do not adequately consider the practical operating conditions for the agents. For instance, most of such works use simulated time, or use inflexible primitive agents (for the definition refer to Section 1.2.1). The use of such agents requires an iterated round-robin execution fashion in the simulation thus eliminating the impact of the concurrent execution of agents actions. There is therefore an unmet need for development frameworks that facilitate the deployment of multi-agent systems, such as robotic systems, which autonomously coordinate their activities in real time. Our work focuses on the development of practical agent-based mechanisms that can be used to facilitate team formation of robotic agents in the context of spatial and temporal constraints. In
order to help ensure that our coordination schemes and mechanisms can be deployed in real situations, we have developed a software engineering simulation framework that enables the testing of our coordination systems in more realistic concurrent environments, before their real deployment in robotic systems. This thesis describes our agent-based coordination module and the associated software engineering simulation framework. The framework will be tested in a cooperative rescue scenario using simulated and physical robots.

1.2 Research questions and original contributions

This thesis contributes to three aspects of multi-agent cooperation and coordination. The research questions that the thesis addresses are described below.

1. How can we design and develop scalable agent-based mechanisms that facilitate coordination among cooperative agents in distributed and concurrent environments? Our mechanism must address the conditions of physical and real scenarios.
   - How can we model concurrent activities of agents in situations where agents need to interact with several agents concurrently?
   - How can the agents deal with uncertainties associated with message delivery?

2. How can we design and develop scalable mechanisms that allow for automated monitoring and management of complex tasks, where each complex task is composed of multiple inter-dependent subtasks?
   - How can we specify complex tasks in a manner that allows agents to identify the subtasks that must be executed concurrently (for instance with respect to coordinated actions)? How do we capture and specify the temporal constraints of complex tasks that specify the order in which various subtasks must be performed?
   - How can we coordinate the execution of complex tasks in a distributed manner?

3. How can we enable the agents to adapt in dynamic situations?
   - What mechanisms can be used to enable the agents to learn about the environmental conditions or performance of other agents?
The above research questions are elaborated in subsections 1.2.2, 1.2.3 and 1.2.4.

1.2.1 Different levels of agencies

In this section, we define three levels of agencies and specify the differences between them. Each level of agency requires certain properties for the agents (for more details about agents and their properties refer to Sections 2.1 and 2.1.2).

Some work in the literature have used agents created by some object oriented language. Such agents are usually entities that encapsulate some state, and are able to execute actions. Agents in those systems have some fields and methods. The fields represent various variables for an agent, and methods are used to change the values of these variables. In such systems agents are executed using an iterated round-robin approach, which simulates concurrent execution of agents. However, such systems cannot be transferred to physically distributed systems due to their central execution mechanism. We refer to these types of agents as primitive agents.

Another level of agency uses primitive agents but further enhances the agents’ autonomy by including a thread (or several threads) for each agent. The thread feature improves agents’ autonomy by enabling them to control their own behaviour instead of being operated by another entity. Such entities have autonomous goal-oriented control over their own execution, and normally communicate with each other using blocking synchronous messages. In those systems, when an agent sends a message then it cannot operate, and must wait until it receives a response from the requested agent. We refer to this group of agents as autonomous agents.

A more advanced level of agency refers to an extension of autonomous agents that communicate using asynchronous messages. The asynchronous message sending further enhances agents’ autonomy by enabling them to continue operating without waiting to receive a response. Such agents are usually equipped with mechanisms that enable agents to deal with situations where message delivery is not guaranteed. In addition, these agents may use interaction protocols and agent communication languages to interact with other agents. We refer to such agents as advanced autonomous agents.

This thesis starts with an implementation of the framework that uses primitive agents.
(Chapter 4), and changes the implementation gradually so it uses *advanced autonomous agents* (chapter 5, 6 and 7).

### 1.2.2 Modelling agents’ concurrent actions

Most frameworks developed for multi-agent cooperation and coordination do not consider realistic characteristics of distributed and concurrent systems (Kraus *et al.*, 2003; Scerri *et al.*, 2005). They mainly use *primitive agents*, and a central manager component that selectively executes their agents. Such systems do not consider realistic concurrent features of real systems and therefore are not able to capture the impact of the concurrent activities of agents.

Our framework employs multi-threaded FIPA-compliant (FIPA, 2000a) agents, where each agent has its own thread of execution and therefore the agents perform their activities concurrently. In addition, a technique based on Coloured Petri Nets’ (CPNs) (Jensen, 1997a) modelling and representational semantics was used to model and control the activities of the agents. We model agents’ extended interaction in terms of agent conversations. The agents decide about what to do depending on the information that they receive from other agents in each conversation. Each conversation has its own thread of execution, thus multiple conversations can be executed concurrently. In this connection, the CPN technique enable us to specify the concurrent activities of the agents.

Another deficiency of most of the simulation frameworks is that they ignore asynchronous message sending. Modelling asynchronous messages that are sent over a network reflects the realistic situations where agents must deal with situations where message delivery between agents is not certain. Agents performing under such conditions may receive their responses after a long period of time due to various reasons i.e., network traffic or overload of the sending agent. Moreover, messages may be lost due to network problems. Therefore, agents must have mechanisms to deal with situations where messages are lost or delayed. Our framework addresses such problems by employing a ‘timeout’ mechanism. Consequently, the agents in our framework use asynchronous messages and respond to their requests in a timely fashion. The framework offers a relatively realistic simulation environment by employing the same concurrent and distributed technology that is deployed in the physical robotic system.
1.2.3 Modelling tasks

Our framework considers complex tasks. Several subtasks compose a single complex task that exist on a higher abstraction level. Temporal dependencies between subtasks establish the relative order of execution between various subtasks in complex tasks. Several formalisms including task trees and Relational Graphs (RG) have been used for modelling task interdependencies (Cruz et al., 2007; Zlot and Stentz, 2006; Gerkey and Mataric, 2002). CPNs are advantageous in this respect, since they allow for modelling concurrency. Our framework employed CPNs to specify the temporal dependencies of complex tasks. The use of task CPNs in our framework facilitates agent management over the order in which various subtasks must be performed. This feature is especially useful when different agent skills are required in different phases of the task. Without such a facility, agents may erroneously recruit extra teammates that may not be required and therefore waste resources.

1.2.4 Adapting in dynamic situations

Any open agent system must have mechanisms that enable the agents to adapt to the dynamics of the environment. In an open agent system, agents may leave the system and new agents may join the system. In addition, the environmental conditions may change dynamically. Any open agent framework therefore must have mechanisms that enable the agents to adapt to changing circumstances. We show how our coordination scheme supports adaptive coordination to altered conditions in dynamic situations. The adaptation mechanisms enable the agents to alter their teammate selection strategies according to circumstances. We will test the adaptation for two types of situations: adopting the best strategy under various circumstances, and adapting to unreliable situations where partner agents do not commit to their contracts.

1.3 Publications

The material from several sections of this thesis has been published in peer reviewed conference proceedings and journals. These publications are as follows:
URL:http://dx.doi.org/10.1109/WI-IAT.2010.260.

URL:http://dx.doi.org/10.1007/s11227-009-0372-8.


• Ebadi, T., Purvis, M.A. and Purvis, M.K. (2008), Partner Selection Mechanisms for Agent Cooperation, In Proceeding of the Third International Workshop on P2P Computing and Autonomous Agents in conjunction with the International IEEE Wi/IAT08 Conference, Sydney, Australia, 554-557. This paper is partially based on the work described in chapter 4.
URL:http://dx.doi.org/10.1109/WIIAT.2008.294.

URL:http://dx.doi.org/10.1109/PDCAT.2008.45.
1.4 Thesis structure

I use the words ‘we’ and ‘our’ in this thesis although I was the only person who designed, implemented and executed the framework. The plural form is to appreciate the guidance and feedback of my supervisors.

Chapter 2 provides a background on agent technology and describes the fundamental concepts related to agents and multi-agent systems. It also describes various mechanisms that can be used for modelling agent interaction protocols. In addition, it introduces several agent platforms and compares their suitability for modelling distributed multi-agent systems. Chapter 3 presents the related work in the area of agent cooperation and coordination and describes the advantages and disadvantages of some current systems. Chapter 4 introduces the basic agent model for our framework. The first part of the chapter employs primitive agents and the second part replaces the primitive agents with autonomous agents. Chapter 5 replaces the autonomous agents with Opal agents and makes use of its Conversation Manager module which allows for higher degree of concurrency in the system. Chapter 6 introduces a model for tasks and demonstrates how agents can adapt in dynamic situations. The framework will be tested on physical robots. Chapter 7 describes the technologies used for the physical robotic experiments. It also explains the experiment that was run using physical robots. Chapter 8 discusses the contribution of this thesis, its limitations and future work.
Chapter 2

Background

Computer technology has gone beyond traditional computational systems that focus on solving fixed problems as a single entity. Computers can be used to solve complex issues that arise in open and dynamic environments where new problems arise and computational entities may appear or be removed from the system. In such environments, heterogeneous systems may interact, form different organisations and cooperate in order to solve a problem. Thus there is a need for software paradigms that can capture these features. Agent technology has been suggested as an engineering approach that enables the systems to operate autonomously in changeable environments and that can adapt in response to changes in the environment. It can be seen as a development in software engineering that promises to address issues like managing complexities in a concurrent and autonomous fashion.

This chapter introduces Agent Oriented Software Engineering (AOSE). It then introduces agents and describes the differences between agents and objects. It also describes various agent’s properties. It then describes multi-agent systems and why they are useful. Moreover, it introduces agent Interaction Protocols and various formalisms that can be used for modelling them. The last part of this chapter introduces several agent platforms and assesses their suitability for our robotics scenario.
2.1 Agent definition

There is no unique universally accepted definition for the term “agent”. Shoham (1997) proposes the following definition that he considers common sense in the Artificial Intelligence community:

“[An agent is] an entity that functions continuously and autonomously in an environment in which other processes take place and other agents exist.”

This definition seems to be the predominant understanding within the community (Shoham partially contributes to this definition). In this definition continuous execution and autonomy are key features in agent systems along with the concept of an environment. Another popular definition in the context of the AOSE is proposed by Wooldridge and Jennings (1995):

“An agent is a computer system that is situated in some environment, and that is capable of undertaking autonomous actions in this environment in order to meet its designed objectives.”

Autonomy in the context of agents refers to the ability of an agent to decide how to act in order to accomplish a goal. An autonomous entity must be able to achieve its goals without direct intervention by third parties (more description on agent autonomy is provided in Section 2.1.2). Such an entity must be able to decide according to various conditions and make the best choices in order to achieve its goals. The agents are able to perform various actions that are available to them. However, these actions normally have preconditions which specify the required conditions in which such acts can be performed. Such a definition of an agent is not comprehensive although it provides some limits in terms of what can be determined as an agent.

A stronger notion of agency suggests that in addition to the features mentioned above, the agents must be developed by using concepts applied to humans such as social ability or mental qualities (Wooldridge and Jennings, 1995; Shoham, 1997). The former refers to the agents’ interactions and cooperation, while the latter refers to the mental components of agents, such as beliefs, capabilities, choices, and commitments. A discussion of numerous agent definitions and classifications is provided by Franklin and Graesser (1997) as well as by Etzioni and Weld (1995).
2.1.1 Comparing agents and objects

There is a clear distinction between defining an object in Objected Oriented languages and agents. However, it is a common mistake to refer to such objects as an agent. This section tries to identify the differences between these two concepts. Since the aim of our framework is to create a realistic multi-agent system, agents properties are of extreme importance and must be defined precisely. In addition, we would like to distinct our work from most other simulation frameworks in the literature which use the term agent to refer to any entity regardless of its characteristics.

OO technology can be extended in various ways to provide properties required for agent systems. Agent-based systems are an extension to OO, as OO is an extension of the modular procedural programming languages (Odell, 2000). It is true that objects can be used to support agent-based systems; now the question is why do we need agents?

Agents and objects both use decomposition, abstraction, and organisational techniques to model complexity. However there are fundamental conceptual differences between agents and objects. A number of studies have defined the differences between agents and objects (Jennings, 2000; Wooldridge, 2009; Silva et al., 2003; Odell, 2000). The main differences are listed below. Our criteria for identifying the differences are based on Jennings (2000) and Odell (2000).

- Objects can only communicate by calling methods on other objects. The OO message broker matches each message to a particular method invocation for a single object. Although the same OO method can be used for agents, agents’ messages can be very rich in terms of their expressiveness, for which an Agent Communication Language (ACL) is required to express communication among agents. The ACL syntax can be defined according to the application requirements (ACL is defined in Section 2.5).

- Agents use asynchronous communication which means the flow of control is unknown from one agent to another. Therefore an agent can initiate actions at any time according to its own discretion.

- Agents may engage in multiple and concurrent interactions by using threads or processes.
In OO languages, objects are confined to the features of their associated classes. However, agents can provide a more flexible approach by allowing dynamic classification, for instance, agents can change their roles or play multiple roles in different interactions.

Objects are centrally organised, as each method of an object must be invoked by another component. While agents can support centralised approaches, they can also provide totally distributed computing services.

Agents often are localised in impact and scope. A single agent that is part of a large system has less impact on the sustainability of the community of which it is a member, for example, an individual bee plays a negligible role in the sustainability of a bee colony. However, if an object is faulty, it stops the whole system. Moreover, agents have a limited scope and make their decisions based on their local perception (for instance, visibility range in robotics scenario).

Agents can show emergent behaviour. The interaction among concurrently running agents may lead to groups of agents behaving as an organic entity at run-time.

The decision on whether to use an OO- or agent-oriented approach for a system must be carefully made according to the requirements of each application. Various factors such as system size, determinism, degree of concurrency, openness and autonomy of its entities must be considered before adopting one or the other approach.

2.1.2 Agent properties

The following defines the main properties of agents. These properties are ordered based on their importance. It starts with properties considering the agent’s fundamental characteristics that feature a strong notion of agency.

Reactivity

Reactive behaviour is the minimum requirement for an agent, and refers to responding to changes that occur in the environment in a timely fashion. Reactivity is a broad term and
can be interpreted widely. Such interpretation may refer to all communicating computational entities. In the context of agency, it may refer to the activation of agent’s behaviour by the environment via use of sensors (intelligent agents are capable of observing their environment) or receiving messages from other agents.

**Autonomy**

Castelfranchi (1995) believes not all agents are autonomous, so he separates autonomy from agent-hood. He suggests that autonomy is a relational concept and uses human analogy to define this notion. He claims that the most significant kind of autonomy is goal-autonomy, which itself is dependent on the “non-negotiability of beliefs”. His definition of autonomy mainly concentrates on what is generally accepted as internal autonomy. Internal autonomy is not concerned about the agent’s independence and addresses issues like how an agent selects its behaviour. Abdelkader (2004) discusses two interpretations of autonomy: self-governance (the agent is directed in choosing goals by a set of motivations), and independence (the agent is independent from other agents). He defines autonomy as self-governance as he believes independence is not a sufficient criterion. Carabelea *et al.* (2004) go even further and identify five types of autonomy: user-autonomy, social autonomy, norm-autonomy, self-autonomy and environment autonomy. They agree with the views of Castelfranchi (1995) and Hexmoor *et al.* (2003) views on the relational property of autonomy.

Hexmoor *et al.* (2003) suggest that autonomy must be considered as a property of agents instead of leaving it to some architecture that guarantees this property. Wooldridge’s (2009) interpretation of autonomy suggests that agents must be able to decide about whether or not to perform an action requested by another agent based on their states and without direct intervention by third parties. He also suggests that agents must have their own threads of control ((Wooldridge, 2009), p.30). Wooldridge uses the term proactiveness (for definition see below) which suggests that agents “exhibit goal-directed behaviour by taking the initiative in order to satisfy their design objectives” ((Wooldridge, 2009), p.27). His interpretation of proactiveness concerns both execution and motivational autonomy which can be captured with the concept of autonomy.
**Proactiveness**

Agents must exhibit proactive behaviour. Proactiveness refers to a property that enables agents to automatically trigger their actions depending on the agents’s states (Bauer, 2002). A proactive agent typically has one or more threads of control.

**Social ability**

Social ability is defined as the agents’ capability for exchanging information with other agents and possibly humans (i.e., communication) (Flores et al., 2009; Wooldridge, 2009). The social ability enables agents to interact in ways that resemble human social behaviours such as cooperation, coordination and negotiation. Some researchers believe that social ability is a property of agents that have an explicit model of their communication partners (Moulin and Chaib-Draa, 1996). This approach is further encouraged by high-level communication languages, such as specified by the Foundation for Intelligent Physical Agents which is explained in Section 2.5 (FIPA, 2002a).

**Adaptability**

This property considers the internals of an agent. Agents use the information provided at initialization time and information that is acquired during execution (agents experience) in order to support their decision processes. Agent interaction with other agents and the environment can lead to changes in the rules that drive agent behaviour. Adaptability refers to changes in agents driving rules in order to adjust agent behaviour according to new circumstances (Floridi and Sanders, 2004). Adaptability is a key feature that enables agents to exhibit intelligent behaviour.

**Situatedness**

An agent performs its actions while situated in an environment (virtual or physical). Agents are able to sense and alter their environments.
Personality

Agents can have personality or character (Etzioni and Weld, 1995). Gmytrasiewicz and Lisetti (2002) emphasise the significance of the role and usefulness of the notions of mental attributes, such as emotions and personality. They believe that agents’ emotional states control the agents’ decisions. In their work any changes in the agent’s emotional state (or mode) alters the agent’s behaviour for making decisions. Castelfranchi et al. (1998) define personality as a combination of traits and attitudes, and use the notion of attitude to present reactive behaviour to manage cooperation strategies in agents. Gratch and Marsella (2005) agree on the crucial role emotions play in human intelligence and the importance of their integration into agents (they refer particularly to virtual agents). They demonstrate numerous effects of emotions on cognition and interaction, that must be known by agent designers since they can impact their applications.

2.2 Multi Agent Systems

From an individual agent’s perspective, Multi Agent Systems (MAS) distinguish from single-agent systems mainly due to the fact that the environment’s dynamics can be affected by other agents. Along with the uncertainty that may be present in the domain, other agents may change the environment in various and unpredictable ways. Therefore all multi-agent systems can be considered to have environments (Stone and Veloso, 2000). The characteristics of MAS are that (1) no agent has the complete information or capabilities to solve the problem and thus has a limited ability; (2) there is no global (or central) control in the system; (3) data are decentralised; and (4) computation is asynchronous (Sycara, 1998). Generally MAS must be able to solve complex problems that are significantly beyond the capabilities of a single agent due to limitations of its resources.

2.2.1 Why use multi-agent systems?

Using multi-agent systems is also advantageous even in domains where a problem can be resolved in a centralised and non-distributed manner. Multiple agents provide parallel computation by means of the operation of multiple entities. This can lead to faster execution of
tasks, for instance, in a particular domain, a task can be decomposed into several subtasks in which each subtask can be handled in parallel by a separate agent. Such domains will greatly benefit from MAS. A good example of such a domains is logistics. In the AAMAS 2006 conference, Steve Benfield from Agentis provided his research finding which shows a significant saving (four to five times) in development cost and time when using agents. He suggests that using agents for solving large and complex industrial systems accelerates the process of finding solutions and helps to bridge the gap between the industrial and research community (Benfield et al., 2006).

Another advantage of parallelism provided in MAS is robustness. A multi-agent system is capable of recovering from faults by replacing the faulty agents. If control and responsibilities are properly distributed among different agents, the system can tolerate failures by one or several agents. In systems where one entity controls the whole system, the failure of such an entity leads to the failure of the entire system.

2.3 Agent oriented software engineering

Building effective software for real-world applications is extremely difficult. This is mainly due to the high number of interacting entities in the systems (Jennings, 2000). Perhaps the main difference between an object and agent is that an object is neither autonomous nor proactive (for more details refer to the Section 2.1.1). Traditional object-based computing considers software systems to be comprised of several “functional” or “service oriented” entities (Zambonelli and Omicini, 2004). In such systems the global architecture of the system is seen as the static decomposition of its functions, and interaction among entities simply represents an expression of inter-dependencies (Bass et al., 2003; Shaw et al., 2002; Shaw and Garlan, 1996). Therefore concepts such as society or entities having multiple roles simply do not make any sense in such systems.

Agent Oriented Software Engineering (AOSE) argues that agent oriented approaches facilitate development (including modelling, design, verification and construction) of complex software systems, and show significant advantages over the classical OO approaches (Jennings, 2000).

Complex systems may consist of interdependent hierarchical structures (Simon, 1996).
The hierarchy can concern temporal dimensions (change over time) and rigid relations (structural dependency) between entities. Booch et al. (2007) suggested that Object Oriented Analysis and Design (OOAD) techniques for managing complexity can be borrowed and used in AOSE. The key techniques are Decomposition, Abstraction and Hierarchy. AOSE replaced the term Hierarchy with Organisation to emphasize the importance of organisation (Jennings, 2000). The following describes these techniques.

- **Decomposition** refers to the process of dividing large problems into sub-problems of manageable size. It shifts the focus to a set of sub-problems as opposed to solving a big problem in one go (Pena et al., 2005).

- **Abstraction** refers to defining simplified models of the system by hiding unnecessary details. The power of abstraction comes from limiting the scope of the design, allowing the designers to concentrate on the core issues.

- **Organisation** refers to identifying and maintaining the relationships between different components of the system.

Agents can be used to decompose complex problems similar to objects. In complex systems it is impossible to predict all possible solutions. Each problem can be decomposed into sub-problems and assigned to agents. Sub-problems are further decomposed and assigned to agents. Agents may not be able to provide all the required functionality to ultimately solve their problem. In such situations autonomy is required to find an entity that can provide such functionality. In addition, the expressiveness involved in interactions brings up the necessity for modelling interaction on a semantic rather than a syntactic level.

Abstraction hides the details of the system from the designers’s view in order to reduce the perceived complexity. Therefore interaction between abstract concepts and their components can be at a very high level. Interaction may represent coordination, cooperation or negotiation patterns. Agent technology has provided standardised methods for dealing with the syntactic level of interaction (for instance ACL). It also provides generalized semantic interaction patterns (for instance Interaction Protocols). These properties allow designers to deal with high-level technical issues and focus on problem domain issues.
2.3.1 Concurrency in multi-agent systems

Social and autonomous characteristics of multi-agent systems demand a high degree of concurrency in agent-based systems. Autonomous agents may be required to engage in social activities like cooperating to perform tasks where all the agents must interact concurrently. The concurrency can refer to both the inter-agent and intra-agent levels. The former refers to the concurrency present on the system level, while the latter refers to concurrent activities performed within one agent. The intra-agent concurrency entails the social aspect of agency, for example, the social aspect must deal with a significant amount of concurrency normally required for conversation and communication between various agents (Tariq et al., 2005).

One of the main signs of a poor agent design is the lack (or low amount) of concurrent problem solving in a system (Wooldridge, 2009). In these systems, agents are executed in a round-robin fashion, in which one agent does some processing, and once it finishes the next agent is executed, and so on (Wooldridge and Jennings, 1998). Those systems are based on a poor design, since there is only one single thread of control. In those systems, concurrency, which is the most significant potential advantage of multi-agent systems, is not exploited. In concurrent systems agents may respond and react to the environment at various levels, and cooperatively solve multiple problems. Hence, one of the main objectives in designing and developing our framework was to provide a reasonable amount of concurrent problem-solving activity. This was also a fundamental consideration in connection with our development of a realistic simulation agent system that performs under concurrent conditions of real systems.

Deyi (2004) introduced an Agent Oriented Programming (AOP) language called A++ for developing agent-based distributed systems for mechanical engineering design. Their language allows for defining tasks as processes (or threads) that run concurrently and have various priority values. Their work violates the autonomy of agents by allowing direct access to agent methods in different tasks.

Tariq et al. (2005) introduced a concurrency model that supports both inter-agent and intra-agent concurrency level for Scalable, fault tolerant, Agent Grooming Environment (SAGE) \(^1\). Their work introduces an Agent Construction Model (ACM) that is based on

\(^1\)SAGE is a second generation FIPA-compliant MAS framework being developed at National University of
event driven multi tasking and works on the basis of a cooperative scheduling concept. They defined a scheduler whose main job is to control the execution of agent tasks. A round-robin execution policy was used for execution. But, their system does not support true concurrency since it is the result of an interleaved execution of the tasks.

Bellifemine *et al.* (2001) used a similar approach to Tariq *et al.* (2005). Their work also considers the cooperative scheduling concept, where a collection of behaviours are scheduled and executed to carry on agent duties. Their work suggests that every agent executes in its own thread, and that each agent has one thread. In their model each agent has an embedded round-robin scheduler that schedules the agent behaviour (all agent behaviours are run within a single Java thread). They used a cooperative scheduling method that does not allow a behaviour to be executed until the previous behaviour ends.

DeLoach (2001) defined the notion of concurrent tasks in order to specify the concurrent behaviours of agents. Each task has its own thread of execution and defines the behaviour of an agent. It also integrates inter-agent as well as intra-agent interactions. Their approach allows agents to choose among different tasks. Their system does not define a thread per agent and therefore does not support the autonomy of agents.

### 2.4 Foundation for Intelligent Physical Agents

As was mentioned in Section 1.2.1, the framework developed in this thesis employs several levels of agencies through the development process and gradually moves towards using *advanced autonomous agents* which comply with standards specified by Foundation for Intelligent Physical Agents (FIPA).

FIPA is an international body registered in Geneva, Switzerland in 1996 that endeavours to develop standards for various aspects of agent technologies that enhances the interoperability of agents by providing open specifications for agents and agent-based systems. FIPA was officially accepted by IEEE as its eleventh standards committee in 2005. When FIPA was established, there were not many commercial agent-based applications. There was therefore a need for a standards committee to promote common specifications for the development of open agent systems.
Figure 2.1: An overview of FIPA specifications, adapted from Willmott et al. (2004)

The FIPA specifications cover several aspects of agent technology. An overview of the areas covered by FIPA is shown in Figure 2.1. Among these topics, Agent Communication Languages (ACL) and Interaction Protocols are described in details (in sections 2.5 and 2.6) since we will refer to these topics throughout this thesis.

2.5 Agent Communication Languages

Agent Communication Languages (ACL) provide agents with a means of exchanging information and knowledge. The ACL deals with the syntax inside a message, and has structures that support participants having a common understanding of the message syntax. This common understanding is achieved with the help of a shared ontology. An ontology is a specification for defining the semantics of the terms used in a message (Labrou et al., 2002). At the technical level, ACL messages are transported over a network using a lower-level protocol (for example, SMTP, IIOP, or HTTP) where the types of messages and their meanings are defined by the ACL. However, this does not mean that agents just engage in single-message exchanges, because agents may have extended conversations (for instance in auctions). More-
over, the agent’s high-level interpretation of strategies and behaviours determines the agent’s communicative (and non-communicative) behaviour. In addition, in the context of the ACL, the speech acts-like statements are represented as *performatives*.

The most popular ACLs are the Knowledge Query and Manipulation Language (KQML) (Finin *et al.*, 1994) and FIPA ACL (FIPA, 2002b). KQML and FIPA ACL are very similar since they follow the same principals. Since our aim is to constructor a FIPA compliant framework, the FIPA ACL is chosen for communication among agents.

### 2.6 Interaction protocols

Interaction Protocols (IPs) can be viewed as a shortcut for agents’ reasoning. They enable agents to provide an appropriate response for a message in a multi-agent system (Greaves *et al.*, 2000). Agents employing an IP must handle the interactions specified in the IP. Therefore all agents participating in a cooperative action must know the IP that is going to be used in the interaction.

Interaction Protocols are similar to communication protocols in distributed systems (Holzmann, 1990). Early work in multi-agent systems adapted communication protocols to facilitate interaction among agents. Communication protocols cannot be used in multi-agent systems due to fundamental differences between agents and processes. One of the main differences is that an agent is an autonomous entity that can form groups to satisfy its tasks. Another difference arises from the complexity of the messages in multi-agent systems, since agents’ messages are richer than messages in distributed systems (Koning *et al.*, 2002). Huget and Koning (2003) worked on the engineering aspect of IPs. Their approach suggests that the design process of an IP must include five phases involving “requirement analysis, formal description, validation, protocol synthesis and conformance testing”.

Several formalisms have been used to model agents IPs. A review of these formalisms is provided in Section 2.8. This thesis heavily uses IPs to model agents’ activities.
2.6.1 FIPA protocols

FIPA provides a set of protocols to model the interaction among agents. These protocols specify the orderings of message types that must be followed in order to accomplish a certain interaction. Figure 2.2 is an example of the FIPA Request protocol represented using the Agent UML (AUML) diagram notation (AUML is defined in Section 2.8.1). The FIPA Request protocol is often used as a simple scenario for testing prototypes. It models the interaction between two agents where an agent asks another to perform some actions. The first message in the IP states the intention (requesting action) of the initiator. The receiving agent either agrees to perform the action and sends an agree message, or refuses to perform the action by sending a refuse message. If the receiver agrees, it must send the result of its action back to the requesting agent. The result of an action can be either failure (in case a failure occurs), inform-done (if the action was performed successfully and does not require the result) or inform-result (when agent was performed successfully and the result must be sent to the requesting agent in the reply message).

2.7 Belief-Desire-Intention (BDI) model

The formal representation of Belief-Desire-Intention (BDI) refers to a specific class of agents that is based on the philosophical model of human practical reasoning developed by Bratman (1987). This section defines this architecture and describes why it is not suitable for our framework.

In each BDI architecture agents have beliefs which represent what an agent believes about the world. The agents obtain their beliefs by sensing the environment. The agents have certain desires which specify the application’s behaviour. Moreover, they select their behaviours in such a way that they proceed towards their goals (intentions). An agent’s intentions must consider a balance between the reactivity and goal-directed behaviour of the agent by committing to plans and periodically reviewing them. The goal of the BDI approach is to be flexible and robust.

The lack of validation tools puts some constraints on the usability of BDI frameworks; for recent developments in planning and validation in BDI refer to (Sudeikat et al., 2007;
Figure 2.2: FIPA Request Interaction Protocol (FIPA, 2000b)
De Silva et al., 2009). In addition, BDI agents rely heavily on considerable amount of reasoning and planning for selecting their behaviours. Thus generally there is some limitation on using the full potential of BDI agents. Georgeff et al. (1999) suggests the BDI model does not suit certain kinds of behaviour. In particular, Georgeff et al. (1999) claims that the basic BDI model seems to be inadequate for developing dynamic systems in which agents must learn and alter their behaviour. They also claim the basic BDI model is not concerned with architectural aspects of agents’ behaviours. Ziparo et al. (2010) suggest that designing agents’ plans using BDI agents is difficult, particularly in systems involving dependency between actions. Moreover, Winikoff and Cranefield (2010) reported that verifying correctness of systems containing BDI agents is very difficult due to the large space of possible behaviours.

2.8 Distributed formal methods for modelling agent interaction protocols

Cooperating agents need to interact in order to achieve their goals. In many cases the interaction is explicit through direct communication among agents. There therefore is a need for standards and formalised techniques that support development of interaction among cooperating agents. Our framework uses a modelling technique to model IPs for communicating agents. In order to justify the use of our selected modelling technique (Coloured Petri Nets), this section provides a survey of various modelling formalisms available for modelling agents’ interaction, and later analyses their suitability for modelling interaction among robotics agents. These formalisms are developed as generic languages that are capable of expressing messages and protocols for inter-agent communication and cooperation. Despite all the advantages in using formalisms, one must note that the choice of a modelling formalism of interactions confines the system in terms of the features that it offers. Thus developers must take into consideration the various characteristics of each formalism.
2.8.1 Agent Unified Modelling Language

The Unified Modelling Language (UML) unifies and formalises the methodologies of various approaches to the object oriented software lifecycle. It supports static and dynamic models. The former includes class and package diagrams that describe the static semantics of data and messages while the latter refers to modelling interaction diagrams, state charts, and activity diagrams. Agent Unified Modelling Language (AUML) is an extension to UML. It is a visual modelling language for AOSE and was standardised by the Object Management Group (Object Management Group, 2003). It comprises a set of UML idioms and extensions to support agent-based application (Odell et al., 2001). AUML is used in the FIPA standards (FIPA, 2002c, 2003). Bauer et al. (2001) used AUML to model agents’ IPs. They introduced protocol diagrams to extend UML state and Sequence Diagrams. Their extension includes agents roles, multi-threaded lifelines, nested and interleaved protocols, extended message semantics, parameterised nested protocols and protocol templates. An agent role describes a set of distinguished properties or behaviours for an agent. The agent lifeline defines the period of time in which an agent exists. An agent lifeline may split into several lifelines in order to deal with parallelism and decisions, for instance, a lifeline can be split into two to handle proposals and not-understood messages. AUML allows for defining nested protocols and interleaved protocols where nested protocols refer to protocols that are defined within another protocol. Interleaved protocols are required in situations where a participant requires more information, for instance, a participant in an auction may request for some extra information before putting a bid. Nested protocols have input and output parameters. The input parameters include interaction threads and messages (that are received from other protocols). The output parameters are the interaction threads that are started from within a nested protocol and are continued outside the nested protocol. Messages that are sent from inside the nested protocol of agent roles are not included in the nested protocol.

Chira and Dumitrescu (2007) proposed a Multi-Agent Design Information Management System (MAIDS) to support the distributed processes. Their goal was to manage information, integrate resources that are distributed across a computer network and facilitate collaborative processes among agents. MAIDS supports cooperation and coordination among agents using direct communication. Each agent in their system belongs to a particular MAIDS.
society, and was designed using the AUML agent class diagram that specifies roles, state description and actions, methods, service descriptions and supported protocols. Moreover, the authors used AUML protocol diagrams to model protocols for multi-agent interaction in their system.

Huang et al. (2003) used AUML to model a community health care system. They used a use-case diagram to represent care service scheduling. In addition, they used interaction and acquaintance models to model communication, interaction and coordination among agents. These models represent agent roles that allow knowledge-level agents to share and exchange resources. Their system facilitates coordination among agents by enabling the agents to read and write domain related information to and from the organisational resource. They suggest that the use of AUML has several advantages including inferential capability, adaptable structure, affordable communication and scalability. Inferential capability refers to the agents’ ability to act according to the specification of higher level models of roles and tasks. Adaptability refers to promoting and redesign of models. The design of interaction models and state diagrams allows for affordable communication. Finally scalability refers to the ability of the system to be expanded.

Tong et al. (2004) used AUML for modelling protocols in Agent Factory Development Methodology (AFDM). In particular, the authors used Sequence Diagrams to model agents interactions, and VIPER (Rooney et al., 2004) as a visual editor to design their AUML Sequence Diagrams. VIPER automatically generates agent code according to each diagram. Tropos (Bresciani et al., 2004) proposes a software engineering methodology and a development framework to model agents systems. In Tropos, the agents code cannot be automatically generated, as that system only provides an editor for designing the protocols. Huget (2002) developed a method for automatic code generation for protocols from graphical representations. In particular, he demonstrates how code can be generated for AUML Sequence Diagrams. Ehrler and Cranefield (2004) developed a tool for AUML Linking (PAUL) that allows for modelling and monitoring agents’ conversations by linking code to AUML Sequence Diagrams. In PAUL, protocols are created by directly editing XML files that correspond to the meta-model. Moreover, it allows for automatic execution of AUML protocols. At the time of publication, agents using PAUL were limited to executing interactions between two lifelines (a lifeline represents a role that can be played by an agent and shows
the associated events of that role for the period of interaction). Padgham et al. (2007) introduced a textual notation for AUML protocols (specifically sequence diagrams). They also developed a tool which produces graphical representation of AUML protocols according to textual notations of the protocols.

AUML are minor modification of UMLs where UMLs are essentially sequence diagrams. While UML-based systems are suitable for human readability and visualization, they are not adequate for automated analysis, validation and verification (Gutnik and Kaminka, 2006). Moreover, complex systems that involve a large number of agents cannot be adequately represented using AUML, for example the system defined by Ehrler and Cranefield (2004) support interaction between only two agents. In such systems, there are many messages that are transferred between different agents. Therefore modelling agents’ interactions, and understanding the behaviour of such systems is extremely difficult. In addition, concurrent activities of large agents systems cannot be expressed using AUML.

2.8.2 Finite State Machines

Generally Finite State Machines (FSMs) have a set of connected global states that describe the behaviour of a system. FSMs can depict the flow of action (or communication) in an intuitive way, and can model a wide range of interactions.

König (2003) developed a layered architecture for facilitating communication among agents in cooperating scenarios. They modelled agents’ conversations and decision processes by using FSMs. Their definition of a conversation is based on the Wooldridge (2000)’s definition of conversation which suggests that a conversation is a structure that models the message transfer among agents ((Wooldridge, 2000), p.80). König (2003) modelled conversation policies as FSMs from the viewpoint of a third party agent that does not participate in the conversation. In their model a speaker and hearer role are required for each message. Figure 2.3 shows an example of a FSM formalism for a simple conversation between two agents, namely agent a and agent b. In this example agent a sends a request to agent b to perform a task. Agent b may accept or reject the request. The conversation selects one of the transitions (accept request or reject request) depending on agent b’s decision. In state S1 agent b accepts the request or rejects the request.
According to Bartolini et al. (2005), most multi-agent systems today use a single negotiation protocol that is usually implemented using FSMs. These protocols are normally hard-coded into the agents, thus establishing static and inflexible environments. Such environments do not allow for flexible and dynamic behaviour of agents and therefore cannot be used for modelling agents’ behaviour under dynamic situations. König’s work (König, 2003) is an example of such constant environments. FSM-based approaches have been used to model multi-robot (and multi-agent) systems, but they are constrained by the limited expressive power of FSMs. Although FSMs satisfy the requirements of many sequential interactions, their expressive power is not adequate for modelling complex interactions, especially those with some degree of concurrency.

Figure 2.3: An example of a FSM formalism for a simple conversation between two agents.

König (2003) also designed a protocol machine that was modelled using a FSM. The protocol machine transforms conversation policies into conversation protocols, and provides agents with the required communication services. When an agent makes its decision, the result of its decision is put into a conversation. The protocol machine of the sender agent then converts the conversation into a message (in the form of a speech act). Once the receiver agent receives the message, it converts the message into a decision. He his their architecture with two agents in a manufacturing scenario, but they did not provide any implementation details of their work.
2.8.3 Dooley graphs

Dooley graphs were introduced by Parunak (1996). He separated agent interactions into several conversation segments. Thus each conversation represents a part of agents models and specifies how agents interact at different phases of interaction. Dooley Graphs consist of nodes and edges. The nodes represent agents and the directed edges correspond to discourses and other actions performed by the agent residing in the source of the edge. Each node in a Dooley Graph represents an agent at a certain stage of a discourse although several nodes can represent one agent.

Wan and Singh (2003) used Dooley graphs for analysing interaction among agents. They particularly focused on “causally linked commitments”. They aimed at determining commitments and capturing causal links between commitments as a means for creating flexible and robust agent models. The authors suggest that the relations among commitments are of high importance, since the lifecycle of commitments depends on the link between commitments and other actions. In addition, any break in the connected commitment can impact all the linked commitments. Huhns et al. (2002) used Dooley graphs for handling exceptions in the context of supply chains. They also used Dooley graphs to create collaboration graphs from the conversations. These graphs were used to generate state machines representing agents behaviours. The Dooley graphs helped the agents to determine the various roles of participating agents in a B2B transaction.

Wan and Singh (2004) provided a theoretical model for Dooley graphs, commitments, and causality with relation to Π-calculus. It allows for definition of concurrent systems in the context of business process modelling. Wan and Singh (2004) used Dooley graphs to provide a separate view for agents, their roles and characters. This feature facilitates the conversations of highly committed agents.

Figure 2.4 shows an example of a Dooley graph. In this example agent A sends a message to agents B and C. Once agent A receives its responses back from agent B and C it then sends a message to agent D. Then the agent D sends a message to agent A. The numbers beside the agents’ names refers to different stages for agents (for example A1 and A2 refer to agent A at stage 1 and 2 respectively).
Dooley Graphs have not been widely used in the area of multi-agent systems. In fact Wan and Singh (2003) and Huhns et al. (2002) are the only studies who have investigated Dooley graphs according to our knowledge. This is mainly due to their poor visual expression compared to other modelling techniques like AUML and FSM. In addition, the Dooley graph formalism does not support concurrency and therefore it is not suitable for modelling agents in situations where multiple agents need to act and interact concurrently.

2.8.4 Petri Nets

Place transitions or Petri Nets (PN) are a well known formalism for modelling concurrent systems. PNs are directed, connected, bipartite graphs in which each node is either a place or transition (Jensen, 1997a). Places may contain tokens. If there is at least one token in every place connected to a transition, then that transition is enabled. Any enabled transition may fire. If a transition fires then one or more tokens are removed from each input place, and one or more new tokens are put into the output places. The total number of tokens generated from the transition to the output places may not necessarily match the number of tokens consumed by the transition. This is because in PNs, tokens represent the state of the model and not objects, and thus do not need to be conserved (Jensen, 1997a).

Petri Net-based solutions for modelling agents’ behaviours have gained interest. In particular, there has been a huge effort in modelling MAS using PN (Ferber, 1999). This is mainly due to the PN’s capability for modelling concurrent behaviour and shared resources. PN offers a more descriptive power than FSM in the sense that the set of PN languages is a
superset of regular languages (languages marked by FSM), and it offers a precise model description. Therefore PN allows for richer models of agents. Moreover, combining PNs does not result in the same state explosion as combining FSM. This makes monitoring the running system easier, since it can be seen visually. In addition, PNs support automatic analysis and formal verification of formal properties of the modelled systems.

Celaya et al. (2009) used PNs for modelling multi-agent systems as discrete-event systems. They modelled the actions of purely reactive agents using PNs. In their model transitions represent the instantaneous actions of agents, and the places represent the environmental states of the agents. Agents interact implicitly by changing each other’s environment. They proposed an architecture that models agents indirect interaction via monitoring changes in the environment. In addition, they analysed the behavioural and structural properties of their models to test the interaction properties. They also tested the deadlock-free property of systems using liveness and boundedness properties of the PN model. Finally, they offered an algorithm that allows for creating PN models from the MAS abstract architecture. They tested their system using two agents, but their model is limited to reactive agents.

Cost et al. (2000) used PNs for modelling conversations among agents. They defined a conversation as “useful means of structuring communicative interactions among agents, by organizing messages into relevant contexts and providing a common guide to all parties”. Their approach used KQML ACL for communication among agents, and they demonstrated their approach by taking an example of a standard representation of a KQML Register conversation and converting into a Coloured Petri Net (Jensen, 1997a). The authors did not provide any implementation details for their work.

Marzougui et al. (2010) has defined an extension of PNs called Agent Petri Net. Their work can be seen as an effort towards filling the gap between PN-based modelling of agents that do not offer direct mapping from the model to implementation. Agent PN can represent autonomous agents and communication among agents while maintaining a relatively intuitive graphical representation. Their model integrates the denotation of graphical representation with executable operational semantics. They formally demonstrated their approach using two agents.

PNs have also been used for modelling agents’ IPs with a special focus on modelling agents’ conversations. Gutnik and Kaminka (2006) compared the suitability of different
modelling formalisms for modelling agents’ conversations. In their approach places represent joint conversation states and messages. They tested their approach for an overhearing scenario. Poutakidis et al. (2002) used IPs for debugging a multi-agent system. They realised the unsuitability of AUML notation for debugging agents interactions and defined a translation from AUML to PN. Their approach uses a debugger to monitor conversations and detect situations where agents do not follow the IPs.

The formal aspects of PNs support precise modelling and analysis of system behaviour, while their graphical representation facilitates intuitive understanding of the proposed solution. In addition, they allow for modelling concurrent activities of agents. Moreover, the modular and hierarchical aspects of the PN models can help in designing solutions for complex systems. These features make PN formalism an ideal choice for modelling agents’ activities in distributed and concurrent systems. Thus we decided to use a PN-based approach to model the activities of agents in our system. In addition, we developed a PN-base mechanism to facilitate coordination among agents.

### 2.9 Agent platforms

Most work in multi-agent systems uses primitive agents. In these systems, agents are mainly reactive and can only respond to external stimuli. These agents cannot exhibit proactive behaviour. In addition, agents in these systems send synchronous messages, meaning that the sender cannot execute any other task before it receives its response.

This thesis has endeavoured to develop a fully distributed and scalable framework that enables agents to make decisions based on the information that they receive from other agents and the environment. It is also open which means agents can join and leave the system at any point during execution. In order to create such a system we have used Opal agent platform. An agent platform takes care of the machinery required for sending and receiving messages. This section introduces a few platforms and outlines their important features in order to situate the Opal agent platform. In order to determine the important features of an agent platform and provide a better understanding of the capabilities of agent platforms, a subset of the FIPA definition of an agent platform is described.
An Agent Platform refers to a physical infrastructure. An AP includes machines, operating systems, agent software support, agent management components and agents (FIPA, 2004). In addition, FIPA suggests a Reference model for the management of agents. The FIPA Agent Management Reference Model shown in Figure 2.5 defines a logical reference model for the creation, registration, location, communication, migration and retirement of agents (FIPA, 2004). It represents a logical model for designing agent platforms and provides interoperability between various agent platforms. The FIPA agents refer to communicative computational processes that exhibit autonomous behaviours. All the agents communicate using Agent Communication Language (ACL). Each agent provides a set of services and has at least one owner. Moreover, agents must have a notion of identity, which is called ‘Agent Identifier’ (AID), that distinguishes the agents in a platform.

The Agent Management System (AMS) manages and facilitates the agents in the platform. The AMS serves as a central directory for agents. It assigns a unique AID to agents upon registration. The Directory facilitator (DF) is an optional component that provides a yellow page service for agents. Agents may register their services with the DF or send a request to the DF asking about the available services offered by other agents. The Message Transport System (MTS) provides means of communication for agents.
2.9.1 Comparison criteria

Agents are building blocks of agent-based applications, thus the choice of agent platform defines the level of granularity which the developers must handle. This is especially important in applications where the concept of agent is explicitly required. Therefore one of the important criteria is the strength of the notion of agency offered by the platform. The means of communication provided by an agent platform is also highly important, since an agent platform provides the machinery required for sending messages. The communication criterion is concerned with available technologies offered by the platform along with agent-related communication standards. Another critical feature that must be considered is compliance with FIPA standards. A FIPA-compliant platform allows its agents to communicate with any other FIPA complaint agents. Such features enable the agents of different platforms to interact and interpret messages in a similar way.

2.9.2 Java agent development framework

The Java Agent DEvelopment Framework (JADE)\textsuperscript{2} is a popular open-source agent platform and a leading FIPA-compliant agent framework. It was developed by Telecom Italia, driven by the need for testing and validation of the FIPA specifications. It facilitates developing multi-agent systems through its FIPA-compliant middleware and via a set of graphical tools that support debugging and deployment of agents. In order to improve the scalability of JADE agents, and to consider the constraints of environments, each JADE agent has its own thread of execution. Each agent has a scheduler that executes the agent behaviour. Thus agents in the JADE platform follow a strong notion of agency.

JADE supports asynchronous communication among agents. All the communication among agents is fully FIPA-compliant. JADE supports the management of complex conversations by providing a set of IPs for common scenarios, such as negotiations, auctions, and task delegation. JADE does not require a particular architecture for its reasoning and a primitive implementation could be considered as reactive, thus the reasoning is left for the developers of agent-based systems using JADE. A new implementation of JADE, called JADEX, is provided to fill this gap (Braubach \textit{et al.}, 2005). JADE has a set of tools that

\textsuperscript{2}JADE agent platform: http://jade.tilab.com/
can be used for debugging and monitoring the agents. It also supports a GUI representation of agents that supports management and run-time analysis of the agents. It further provides support for partially or fully mobile agents.

2.9.3 Madkit

Madkit\(^3\) is an open source agent platform that was originally developed by Gutknecht and Ferber (2000) at the University of Montpellier. It strongly relies on the notion of organisation in MAS and follows the Agent/Group/Role reference model: an organisation is formed by groups of agents and each agent plays a particular role. The focus is thus on the organisation and the structural relation between agents in contrast to modelling individual agents as in JADE.

Agents in Madkit can be viewed as active communicating entities which play roles in groups. In Madkit, the behaviours associated with roles are directly implemented on the agent implementation level (part of the agent structure), thus impacting the re-usability and modularity of the platform. Agents in Madkit follow a weak notion of agency, since agents do not necessarily imply proactiveness and their life cycle can be controlled by their organisations. It supports asynchronous communication among agents, either by passing messages directly to another agent or to a higher-level system that sends messages to the agents playing a particular role in a certain group. Madkit uses Java TCP sockets to implement its communication. In addition, it supports the integration of an open set of protocols for connection handling.

Madkit does not support advanced reasoning and that consideration is left to the developers. This is because Madkit was developed to support a scalable framework that can handle a large number of light-weight agents. Since the focus of Madkit is on organisation, a reasoning agent may be implemented for an organisation. It is based on an open agent concept and supports scheduling or threading for agents. Some of the weaknesses of Madkit include lack of FIPA support, low-level communication protocols, and lack of a reasoning engine.

\(^3\)Madkit agent platform: http://www.madkit.net
2.9.4 Cognitive agent architecture

Cognitive agent architecture (Cougaar\(^4\)) is an open-source, Java-based agent framework (Helsinger et al., 2005). It was produced as the result of a DARPA research (Advanced Logistics Program) in which the main goals were to develop a highly scalable and reliable multi-agent platform for distributed systems. Cougaar supports managing agents applications while abstracting the underlying architecture from the developers. The robustness and reliability characteristics of Cougaar has led to a growing community (Haack et al., 2004).

Cougaar agents participate in cooperative activities to solve complex problems in a distributed fashion. Each Cougaar agent involves related functional modules that are responsible for dynamically producing the solution in terms of the problem parameters, constraints, or execution environment change. The agents communicate with one another via a built-in asynchronous message-passing protocol. A Message transport layer provides “link protocols” such as loopback in-memory transport for internal communication, RMI, CORBA, HTTP, SOAP and JMS. Cougaar agents do not support a strong notion of agency due to the lack of support for the use of high level agent communication languages, and because this conflicts with the key features of Cougaar, which are high performance and scalability. Finally, Cougaar agents do not support FIPA standards, since their design objective is to ensure scalability and performance.

2.9.5 Janus

The Janus project is the result of collaboration between the Palermo department of the ICAR institute of the Italian National Research Council (CNR), and the SeT laboratory of the University of Belfort-Montbéliard (UTBM), France\(^5\). It is a relatively new multi-agent platform specifically designed for implementation and deployment of holonic multi-agent systems. It integrates holonic and agency concepts, where the holonic aspect focuses on modelling the organisational aspects, while the agent aspect focuses on modelling the agents as entities composing the system. Janus follows an organisational approach by using the notion of role and organisation as first-class entities (Gaud et al., 2009; Cossentino et al., 2010).

\(^4\)Cougaar agent platform: http://www.cougaar.org/
\(^5\)Janus agent platform: http://www.janus-project.org/Home
A holon can be composed of other holons as sub-structures. This hierarchy in holons structure is referred to as holarchy. A holon can be either an autonomous (or atomic) entity or composed of a group of interacting holons. Janus provides a direct implementation of important features used in the design phase including organisation, group, role, and holon, and capacity to facilitate the transition from design to implementation phase. It includes the notion of HeavyHolons and LightHolons. The former refers to an agent that has its own thread of execution, while the latter refers to agents that do not have their own thread of execution. A Janus agent can play multiple roles in several groups simultaneously. The Janus architecture respects FIPA’s reference architecture but does not support FIPA ACL yet. The Janus supports communication among agents depending on their roles. The role provides a way for communicating with other agents in a particular context. Roles therefore can be considered as the basis for all interactions. In order for agents to communicate, they must belong to the same group. If the group is distributed between a number of kernels, an instance of the group must exist in each kernel. This assumption is not possible in certain scenarios, including the robotics context where each agent resides in a separate robot. The communication method of Janus is a clear disadvantage for the platform for our purpose, and does not suit physically distributed systems.

2.9.6 Opal

The Otago agent platform (Opal) is an agent tool-kit that is implemented using the Java language (Purvis et al., 2002). This project originated at the Software Engineering and Collaborative Modelling Laboratory (SECML) of the Department of Information Science at the University of Otago. Opal is a multi-level agent platform that offers a strong notion of agency. The key idea in Opal is to create an agent-oriented decomposition that meets the criteria of AOSE to model distributed applications (Purvis et al., 2002). Opal is fully compliant with FIPA specifications, thus satisfying the important requirements via providing components such as Agent Management System (AMS), Directory Facilitator (DF) and Message Transport System (MTS). Opal also includes micro-agents that represent the most primitive level of agent instantiation, and perform tasks such as dispatching messages, managing con-

6http://secml.otago.ac.nz/
versations, and executing plans. An important feature of Opal is the support of the FIPA ACL for agent communication. It also provides all basic Java Agent Services (JAS) and supports RMI, FIPA, HTTP and IIOP as Message Transport Protocols (MTP).

Opal agents have the capability to pursue advanced reasoning by employing advanced reasoning engines (Wang et al., 2005). Opal modularity allows for flexible uses. One extension module is the Conversation Manager (CM). The CM is in charge of executing complex conversations between agents (more information about the Opal agent platform is provided in Section 5.2). We decided to use Opal as the agent platform for our framework, since it is FIPA-compliant. This feature enables our agents to communicate with other FIPA-compliant agents using FIPA ACL. The use of the CM and its execution engine, JFern (Nowostawski, 2000), allows for modelling and execution of concurrent activities of agents using Coloured Petri Nets (for more information about Coloured Petri Nets refer to Section 5.1.1). In addition, access to its source code enabled us to easily modify it according to the requirements of our application (the details of these modifications are provided in Section 5.2.1).

2.10 Summary

This chapter has introduced fundamental concepts of multi-agent systems. It has briefly defined agents and specified why agents are different from objects. Various properties of agents, including reactivity, autonomy, proactivity, social ability, adaptability, situatedness and personality, were also described. It described the advantages of AOSE over OO approaches. The AOSE approach offers a high degree of concurrency required in multi-agent systems. This chapter also introduced the FIPA standards, and Agent Communication Languages (ACL). An ACL provides a common framework for communicative interactions so the agents can interpret the message that they exchange. In addition, Interaction Protocols (IPs) were defined. IPs support more efficient strategies for agents in resource-bound environments. They help agents determine the appropriate response for various requests in situations where agents are involved in several interactions over various tasks.

We have also introduced various formalisms that have been used for modelling agents’ IPs. Such formalisms include AUML, FSMs, Dooley graphs and PNs. Among all these formalisms only PNs support concurrency required in modelling complex concurrent sys-
tems. The last part of this chapter has described several agent platforms. We compared these platforms with regard to their suitability for use in fully distributed concurrent systems. We chose Opal as the agent platform for our framework due to availability of its source code and the fact that we were able to modify it according to our needs.
Chapter 3

Related Work

Cooperation is a key issue in MAS. Cooperation refers to the social ability of agents to work together towards a common goal. It is a complex behaviour since it refers to achieving a common goal among autonomous agents in which each agent is self-interested and has its own goal(s). Cooperation is required in situations where agents face complex problems that require different expertise that must be provided by various entities. It is also advantageous in situations where a single entity may be capable of performing a task, which must be coordinated with other agents. In such situations, cooperation allows for parallel execution of multiple tasks, thus decreasing the time required for task performance.

Our framework studies the problem of cooperation and coordination among robotic agents in the context of a “rescue” operation. It includes a number of complex tasks that cannot be performed by a single robot. Therefore robots need to form a team to perform tasks. Each task is composed of several subtasks in which each subtask may be assigned to one or more robots. In addition, our framework includes interdependency between various subtasks. The interdependencies specify the order in which various subtasks must be performed. A Coloured Petri Net formalism was used to represent interdependencies between various subtasks. Some work in the literature has addressed similar problems to the ones addressed in our framework. This work is generally classified as task allocation mechanism. In addition, agents in our framework use multiple criteria for selecting their teammates. This is similar to some existing multi-criteria decision making approaches. Therefore the first part of this chapter describes various task allocation mechanisms including coalition formation
and multi-criteria decision making techniques.

Our framework uses Coloured Petri Nets to model agents activities. Among all the formalisms available for modelling agents’ activities, Petri Nets and Finite State Machines are the most common. Therefore we also describe some work that has used Petri Nets and Finite State Machines for modelling agents’ activities.

The last part of this chapter provides a brief discussion of the limitations of the existing work and justifies our approach.

### 3.1 Task allocation

Task allocation addresses the problem of allocating tasks to a group of agents in uncertain environments. It focuses on developing methods that allow a team’s goal to be decomposed into several sub-goals. It also concerns mechanisms that assign sub-goals to team members in order to solve the overall goal. Task allocation is NP-hard, meaning that identifying the optimal solution is not computationally feasible especially for large problems.

Market-based approaches have been used to facilitate task assignment to a group of agents using contract negotiation. In such approaches, agents act as self-interested entities participating in a virtual market economy. A manager agent plays the role of the seller and offers tasks to other agents playing the contractor role. For each task, a manager auctions the task by offering the task to contractors for a fixed price. Upon receiving an offer a contractor processes its offer and selects the offer that maximises its utility, and then they send a bid to the manager for executing the task. The manager must have a mechanism for selecting the best bids and assigning the tasks to the winner of the auction. A task may need more than one contractor, and the winning contractor(s) must then perform the task. Normally, in auction-based mechanisms, agents can dynamically choose their roles which is either that of manager or contractor.

Central market-based mechanisms have been used to address the problem of coordination and cooperation in multi-agent systems. A popular mechanism is called a “combinatorial auction”. It is used for coordination of a large number of distributed problems. In “combinatorial auctions”, all agents send their bids for a combination of goods or services to a central auctioneer who determines the winner. CASS (Fujishima et al., 1999) and CABOB
(Sandholm, 2002; Sandholm et al., 2005) are examples of such centralised mechanisms. Another popular centralised mechanism is called “reverse auction”, which refers to a centralised market-based mechanism in which sellers send their values to a central coordinator. The coordinator then selects the optimal allocation. Vickrey-Clarke-Groves (VCG) is perhaps the most popular centralised market-based mechanism (Mas-Colell et al., 1995). VCG has two interesting economic properties: it offers an efficient mechanism for finding the cheapest sellers, and it ensures that the joint plan is far better than individual agents’ plans. But centralised approaches are non-scalable and computationally expensive. Moreover, since one entity makes the decisions for the entire system, they are normally slow. Another important weakness of centralised approaches is their susceptibility to failure due to having a single point of coordination for the entire system. This means that the whole system cannot operate if the central entity fails due to any reason.

Shehory and Kraus (1998) designed a heuristic-based algorithm for task allocation via agent coalition formation. They developed an open agent environment by employing an open and distributed agent platform called RETSINA (REusable Task-based System of Intelligent Network Agents, (Sycara, 1998)). In RETSINA each agent has its own thread of execution. Each thread simulates the task queue of an agent along with its available capabilities. When a task is sent to an agent, it decomposes the task into a set of subtasks by using a predefined task decomposition library. The agent must then contact a matchmaker in order to find agents with the required capabilities (all the agents joining or leaving the system inform the matchmaker about their capabilities). The agent then computes a coalition value for each subtask by considering all possible coalitions, and then selects the coalition with the highest value. Once the subtask is completed then the subtask is removed from the agent’s list of tasks. Their algorithm constrains the coalition space by limiting the coalition size to a fixed number. The use of matchmakers is not practical in some distributed robotic scenarios, since it requires the presence of a matchmaker in every region. It also requires agents to constantly update the matchmakers as they move from one location to another. Shehory and Kraus (1998) used several Sun workstations to test their approach.

Scerri et al. (2005) introduced Low Approximate Distributed Constraint Optimisation (LA-DCOP), a task allocation mechanism. It assigns a variable to each agent in which the variable corresponds to agent tasks. Since each agent can execute multiple tasks at the same
time the variable can take multiple values simultaneously. The task allocation mechanism creates a token for each task. The agent that receives a token assigns values to its variable or passes the token to a random team member. An agent’s decision on whether to assign values (tokens) to its variable is based on the team’s best interest. In order to include the opinion of the teammates the LA-DCOP uses a threshold mechanism. LA-DCOP calculates a threshold for the minimum capability that an agent must have before it can accept a task. This threshold is attached to a token and if the agent’s own capability is less than the threshold, it passes the token to another teammate. Thus the optimum allocation depends on a good threshold calculation. LA-DCOP also includes task interdependency, specifically the AND constraint, where the AND constraint implies that the team receives a reward if all subtasks are executed concurrently. The tokens for all AND constraints are assigned to one team member in which it sends a small number of potential tokens. Each receiving agent agrees to accept the token only if a potential token for each of the other real tokens is accepted by another agent. When the agent sending the real token receives the first potential token, it locks up the group and sends the real token in addition to the list of the other agents that have accepted the other real tokens.

Some works have focused on solving the task allocation problem in networks. Sander et al. (2002) proposed a distributed algorithm for task allocation in situations where agents and tasks are geographically dispersed. The goal of each agent is to maximise its own number of performed tasks. Each agent has a limited visibility range that constrains the information of nearby agents and tasks. Sander et al. (2002) adopted a triangulation planar, a computational geometry technique, to determine adjacency information for agents (each node in their model represents a task or an agent). The authors used a weight system that assigns a weight to each task. A task is completed if the number of agents is equal to the weight of the task. Tasks with high weight value require a high number of agents. To address this issue Sander et al. (2002) used a propagation method that informs other agents in addition to the agents adjacent to the task, which suggests a local communication between adjacent agents. Their system involves minimal coordination among agents.

Abdallah and Lesser (2005) adopted a network in which each agent must be connected to a mediator. A mediator receives tasks that dynamically appear at different locations (mediator location) and have connections to other agents (the mediator only interacts with its
connected agents). It is also in charge of decomposing the tasks into subtasks, and assigning them to the agents. However, they assumed that subtasks are independent. Abdallah and Lesser (2005) focused on improving the decision modelling of the mediators. They extended the Concurrent Action Model (CAM) (Rohanimanesh and Mahadevan, 2003) to model a mediator who can mediate multiple tasks in parallel. Weerdt et al. (2007) adopted a social network to solve task allocation using a decentralised algorithm. Their algorithm is based on the Contract Net Protocol (CNP) (Smith, 1998). All these task allocation mechanisms that are based on a network, aim to find the most efficient task for each agent and do not consider the coordination issue in depth.

### 3.1.1 Coalition formation

Coalition formation algorithms have been designed to address the problem of coordination and cooperation in multi-agent systems. They enable a team of agents to use their capabilities in order to maximise their global objective. Coalition formation algorithms allow for a number of teams to be formed in which each coalition can have a value, and the value of each coalition depends on the agents’ capabilities and limitations.

Kraus et al. (2003) considered a coalition formation approach for solving complex problems using agents. Their mechanism consists of a protocol and set of strategies. Their protocol is an extension of the auction mechanism and includes a central manager and multiple agents. The manager supports an auctioneer role and a coalition negotiation manager. The auctioneer sends a Request For Proposal (RFP) that has an associated price, and determines the winning coalition. Each auction in the system is divided into several rounds. In each round, the manager allows for negotiation actions in which each agent either sends a proposal for joining a team, or receives such a request proposal. Each agent can only perform one of these actions in each round. Kraus et al. (2003) also proposed some heuristics that provide a means for deciding on the best coalition to join to. These heuristics required agents to evaluate all possible coalitions, assuming that agents could observe all possible tasks and all other agents. Such assumptions are however impractical in real systems.

The coalition mechanism developed by Kraus et al. (2003) does not consider individual agents’ capabilities after a team is formed and assumes that the agents’ capabilities are
available collectively after a team is formed. They assume once a team is formed then the capabilities are redistributed among team members. These assumptions cannot be made in robotic scenarios where robots’ capabilities correspond to robotic equipment such as excavator buckets or sensors such as cameras. This issue shows that simply collecting adequate resources does not suit multiple-robot coalition scenarios.

(Abdallah and Lesser, 2004a,b) focused on using organisational structures to address the problem of forming coalitions. They introduced a distributed polynomial-time algorithm to optimise agents’ decision-making. Their model builds a hierarchical representation consisting of manager and leaf agents. In order to perform each subtask, an agent may ask other agents in the lower levels of the hierarchy to fulfil the task requirements. The organisation is updated at run-time by applying distributed learning based on task allocation patterns, to improve the global utility.

Wanyama and Far (2006) modelled the agents’ process of coalition formation in addition to modelling the agents commitment level (to their coalitions) in the context of Group-Choice Multi-Criteria Decision Making (GCMCDM). Their mechanism allows for employing multiple coalition formation algorithms. They particularly tested their model using two coalition formation algorithms, one considering similarities between agents’ predilections, and the other one based on ownership criteria. Each agent ranks the possible solutions based on the suitability of each solution for satisfying the combined preferences of the coalition members from the perspective of a given coalition member. The Wanyama and Far (2006) algorithm supports agents’ multi-level commitments to different coalitions.

ARAMS is a multi-agent coalition formation approach (Soh and Li, 2004). It allows a team of agents to dynamically allocate CPU resources to individual team members agents where events dynamically appear in the environment. The agents must assign events so they avoid CPU shortage crises. In order to meet the objective, ARAMS uses a goal-directed coalition formation model. Goal-directed coalition formation allows the agent to seek help from other agents when possible. The agents use learning to improve coalition success rate.

ALARM is an approach that uses reinforcement learning (Sutton and Barto, 1998) and case-based reasoning (Soh and Tsatsoulis, 2001). Reinforcement learning improves the success of coalition formation, whereas case-based reasoning impacts the agents negotiation. In order to facilitate the learning mechanisms, each agent stores each negotiation task. It also stores
each neighbour as a vector. Each case in the case-based learning records the task problem, solution and utility. Therefore each agent has a history of its previous interactions with other agents. The reinforcement learning enables the agents to estimate the potential utility of a possible candidate based on the sum of the utilities of its past interactions and its ability to solve the current task.

3.1.2 Task allocation mechanism for robots

Hoeing et al. (2007) used a prototype system called COMSTAR (COoperative Multi-agent Systems for automatic TArget Recognition) to solve the task allocation problem for autonomous agents in situations where each agent represents an unmanned aerial vehicle (UAV). It assumes that computational resources of multiple robots are required for completing a task. Thus a team of robots must be formed for executing each task. The authors used an auction mechanism to form teams. Each UAV that finds a task starts an auction by announcing the task location and requesting bids from other agents within its communication range. Each UAV may send a bid for a task in which the value of the bid is based on its distance from the task location. Then the auctioneer selects the top \( n \) bidders as winners, and sends a message to the bidding agents and informs them about the outcome of the auction. The winning UAVs must then start performing the task. Each UAV that visits the task leaves pheromone at the task location. A task is considered incomplete if the pheromone deposit at the task location is less than a certain threshold value. If the last UAV visiting the task observes that the pheromone value is less than the expected value, it starts another auction.

Kalra and Martinoli (2006) used the event-handling domain as the framework to study the problem of task allocation using robots. Such domains consider dynamic situations where events occur at unknown times and locations throughout the environment. Hoeing et al. (2007) used an auction-based mechanism to assign tasks to agents. However, their work assumes a simple scenario in which a task is indivisible and can be performed by one agent. Both Hoeing et al. (2007) and Kalra and Martinoli (2006) only tested their systems on a simulation framework and did not test them using physical robots.

MURDOCH is a system that is based on the Contract Net Protocol and auctions designed to address the challenge of task allocation for a group of robots (Gerkey and Mataric, 2002).
MURDOCH is one of the first attempts to apply auction-based methods for coordination of multiple robots that work on several tasks. In contrast to traditional point-to-point message delivery in robotic systems, MURDOCH uses broadcast messaging which means that a message is sent once, providing the system with potential savings in bandwidth. Robots move from one location to another meaning that the robots may move in and out of communication range. To address such dynamic issues, the communication layer of the robots communicate anonymously by means of broadcast.

Gerkey and Mataric (2002) have tested their system for a loosely coordinated box-pushing scenario with three physical robots. Two robots were in charge of pushing a box while the other robot, called "watcher," oversaw the task. The "watcher" robot was in charge of detecting failure in any of the pushing robots and informing the other robot about the failure. If one of the robots fails, then the working robot must continue the task until it is complete. In MURDOCH, the criteria for determining the suitability of the participating robots is hard-coded. MURDOCH uses a hierarchical task structure to represent the agents’ tasks. A task is represented using a tree that can include other tasks (a root node of a tree can be an intermediate node in another task tree). A task tree is a distributed data structure with a declarative representation that describes a complex task. Gerkey and Mataric (2002) focused on the problem of allocating tasks to a group of robots and delegated the definition of task trees to the designers.

Zlot and Stentz (2006) extended the market-based approaches by introducing complex tasks. Their work allows for modelling task interdependencies using task trees. In their system, the agents auction the task trees instead of a single task. The winner of the auction must execute the entire task (i.e., all the subtasks) or may delegate subtasks to other agents by means of subcontracting. Interdependent tasks can be represented as leaf nodes, since they cannot be further decomposed. Zlot and Stentz (2006) used logical operators to represent interdependencies in situations where tasks can be decomposed into subtasks. Subtasks are related by logical operators \textit{AND} or \textit{OR}: an \textit{AND} operator defines a situation where all the subtasks must be performed to satisfy the parent task, and an \textit{OR} specifies that at least one of the subtasks must be completed to satisfy the parent task. This task model is suitable for problems that require loose coordination. More complex operators are required to enable the robots to deal with problems that require tight coordination. Figure 3.1 shows an example
of a task tree for collaborative surveillance using task trees. In this example, the area that
needs to be monitored is divided into two regions (region 1 and region 2), and each region
can be monitored with several robots through use of various Surveillance Points (SP). In this
example, monitoring the region requires monitoring either region 1 or region 2. Solid lines
represent the AND operators and dashed lines represent the OR operator.

Figure 3.1: An example of a task tree for monitoring two regions (adapted
from Zlot and Stentz (2006)).

Dias (2004) adopted TraderBots, an auction-based coordination mechanism, in a multi-
robot space application. The TraderBots allows robots to plan their actions locally. In addi-
tion, it allows for opportunistic centralised planning, meaning that leaders can be formed to
plan for a subgroup of robots and improve the quality of the solution. In this work robots are
self-interested agents, primarily aiming at maximising their own goals. In the TraderBots ap-
proach, each robot has RoboTraders that are the market component of the robot. In addition
to that, their architecture introduces another component called OpTraders. OpTraders are
similar to RoboTraders but they are not associated with robots. Instead, they provide a user
interface and act on behalf of users. The OpTraders are responsible for decomposing tasks
introduced into the system by users, and auctioning them. Once the task is auctioned, then
a team of robots must perform each task. The TraderBots approach allows robots to auction
their committed tasks to other robots if this further advances the robots’ profit. The authors
realise that the frequency of such action impacts the efficiency of their system and only al-
allows for sub-auctioning tasks that have not been scheduled for execution. In addition, their system enables the robots to participate in multiple auctions without being concerned about complex bid-estimation by requiring that bidders anticipate their chance of winning bids. This does not impact the quality of the solution due to frequent re-auctioning of bids, and the dynamic nature of the environment that impacts the cost estimation for bids. Moreover, they demonstrate that increasing the number of offers in each auction decreases the total time for finding the solution. They also mention that this reduces the quality of the final solution. To limit the number of bids for each robot, Dias (2004) suggested a clustering approach for multi-task processing.

Sariel et al. (2007) introduced a task allocation mechanism that addresses the allocation of complex and interdependent tasks to a team of robots that operate under dynamic conditions. Their approach considered precedence constraints on subtasks. The authors used task graphs similar to Vig and Adams (2006) to represent interdependencies among subtasks. In their system, tasks are given to the robots at initialisation time, and the robots form a schedule for theirs tasks. Their robots consider real-time issues, like path planning; thus they update their schedules according to the information received during execution. They used their Dynamic Priority-based Task Selection Scheme (DPTSS) to assign tasks to a team of robots. The main purpose of DPTSS was to incrementally assign tasks by considering the constraint on subtasks and resources whenever a new task must be allocated. They tested their approach using Webots (which is a simulation platform for robots (web, 2011)), and real Khepera robots (Khepera, 2011). The simulation experiments were executed using pick-up and delivery. The real robot experiment was performed using three Khepera robots in a box-pushing scenario.

Landén et al. (2010) also used a task tree for representing a complex task in a multiple UAV scenario. Each node of a task tree corresponds to a task and has certain requirements that must be satisfied. Nodes may represent actions or goals where each action can be a complex action. Primitive actions are represented in leaf nodes, while a composite action is an interior node. Action nodes are executable, whereas goal nodes require plan construction before execution. Each generated plan then becomes a new task tree. Nodes are connected using lines, and can be added or removed at run-time.
Figure 3.2: An example of a task tree for concurrent scanning of four fields Field$_A$, Field$_B$, Field$_C$ and Field$_D$ and then flying to a destination Location$_D$, adapted from (Landén et al., 2010).

Figure 3.2 shows an example of a task tree. In this example, first Field$_A$, Field$_B$, Field$_C$ and Field$_D$ must be scanned concurrently. Once the scanning is performed, then a UAV can fly to Location$_D$. Nodes N3, N4, N5, N6 and N7 represent primitive nodes. Nodes N1 and N2 represent concurrent (C) nodes and node N0 represents a sequential (S) node. In this example, each node is associated with temporal parameters that specify the time period in which a task is expected to be performed. In addition, each node can have node constraints that constrain values of the node parameters. A task tree can also have tree constraints representing precedence, dependence, and organisational relations between the nodes. A constraint network can be derived by considering the node parameters and task tree constraints. Therefore the value for these parameters constrains the network as well as the degree of autonomy of an agent. Figure 3.3 demonstrates the constraint network defined for the task tree represented in Figure 3.2. The authors used an algorithm that recursively finds the potential tasks to be executed in a distributed manner and used distributed constraint satisfaction techniques to determine the satisfaction of the constraints. Landén et al. (2010) did not provide any implementation for their work.
Dahl et al. (2009) used a task allocation method that requires no direct interaction among robots. Their approach suits situations where direct communication between robots is impossible. In Dahl’s work information is communicated between the robots indirectly through use of stigmergy. They developed a Vacancy Chain Process (VCP) inspired by Chase et al. (1988). A typical example of VCP is where a senior staff member of a company resigns. Such an action creates a vacancy for a less senior staff member to fill the position, which then creates. It also creates a second vacancy to be filled. The authors used VCP to address the engineering issues related to modelling complex situations as well as modelling interaction among team members. They tested their system in the context of transportation where robots must move in the environment in order to transport items between different locations. They tested their system using the Player robot device server and the Stage simulator (Gerkey et al., 2001).

Coalition formation mechanisms for robots

Vig and Adams (2006) suggested a coalition formation mechanism for robots in scenarios where the coalition must be formed under certain constraints (a precedence order exists between tasks). They used a high level planner (possibly GraphPlan (Blum and Furst, 1997)) to develop the structure of tasks. A task structure was later used to develop a graph which
represents the interdependencies between tasks. Interdependent tasks in their system were performed using overlapping coalitions. Figure 3.4 shows a precedence order graph for a set of tasks. In this example, tasks $T1$ and $T2$ are independent while task $T3$ cannot be completed before $T1$ and $T2$ are performed. They tested their system both using a simulation environment and physical robots.

![Figure 3.4: An example of a task precedence order graph for three tasks.](image)

STEAM is a framework that models collaborative behaviour of agents (Tambe, 1997). It is based on the theory of *joint intentions* proposed by (Cohen and Levesque, 1991) and the theory of shared plans (Grosz, 1996; Grosz and Kraus, 1996). Agents in STEAM monitor the behaviour of participating agents as well as monitoring the team’s performance. It facilitates the formation of teams based on a hierarchical role-based team organisation, a (hierarchical) description of team, and individual agents activities. STEAM is an effort towards flexible models of cooperation by enabling agents to reason autonomously about coordination and communication in teamwork. STEAM has been used for simulation of various applications such as military robots but has never been used on physical platforms.

BITE is a distributed architecture built for automation of cooperative and collaborative activities in physical robots (Kaminka et al., 2007; Kaminka and Frenkel, 2007, 2005). In the BITE architecture, the task related behaviours are separated from social behaviours. The task related behaviours refer to behaviours controlling the robots’ interaction with their tasks, while the interaction behaviours specify the behaviours that control social activities of agents (i.e., interaction with other agents). Along with a task (or subtask) behaviour graph to manage teamwork, it maintains an organisation hierarchy. In addition, BITE provides a library of hierarchically linked social interaction behaviours. These behaviours are used for im-
implementing IPs that are employed interchangeably, to facilitate selection of task behaviours among team members.

ALLIANCE is a distributed, behaviour-based architecture that facilitates fault tolerant cooperation among a team of mobile robots (Parker, 2002). It enables robots to constantly monitor the sensory feedback of the tasks that can be executed by an individual robot. Such a mechanism enables the robots to adapt their actions according to the current environmental conditions and the actions of other teammates. In ALLIANCE robots dynamically assign and reassign themselves to tasks, in case of failures in their own system or those of their teammates while they collaborate. ALLIANCE uses fixed teams, and robots are added and removed from a team by human interventions. In addition, they use fixed IPs and do not allow for altering the IPs underlying the synchronisation and allocation.

ALICA is a framework for facilitating formation of teams among robots for solving complex problems. It represents agent behaviour by using a hierarchical state machine which consists of \( n \) hierarchical levels. ALICA reinforces reuse of the modelled behaviours as common decisions based on an agent's knowledge of the environment (that leads to a change in its behaviours) that can be made on higher levels of the hierarchy. ALICA explicitly considers team behaviour in its design. It provides a global view of the team while modelling its behaviour. ALICIA has a Behaviour Engine that controls the execution of basic robot's behaviours such as sending messages to robot actuators indicating what robots must do (e.g., the robots must move or kick an object). The Behaviour Engine is a two-level hierarchical state machine.

### 3.1.3 Multi-criteria decision making

Multi-criteria decision-making mechanisms enable agents to select appropriate teammates based on multiple criteria representing quality of agents. Such a mechanism decreases the risks associated with cooperation since agents will consider several factors impacting an agents performance when selecting teammates. This thesis employs a multi-criteria decision making mechanism for cooperative task allocation.

In our framework, agents consider multiple criteria for selecting teammates similar to multi-criteria decision making mechanisms. Agents include time, distance and quality of
various teammates in their decisions. The consideration of multiple criteria enables the agents to make better decisions when selecting teammates. This improves agents’ reward and consequently the performance of the overall system. In addition, agents may need to value certain dimensions higher than others according to various circumstances. Agents thus may be required to change their values for different dimensions in dynamic situations.

Gujral et al. (2006) used multiple dimensions for agents. In their systems, they modelled multiple factors when selecting team members. The authors suggested that modelling multiple factors improves agents’ rewards. Griffiths (2005) used a multi-dimensional trust mechanism to select appropriate agents to cooperate with. The authors define trust as a “means of assessing the risk of cooperating with others”. Agents may fail at performing tasks, or succeed but at a lower quality or higher cost. Griffiths (2005) refers to such features as dimensions of trust. Thus the trust value of an agent must specify the probability that an agent commits to its contract. They used four dimensions for trust in their work, including success (probability for successfully completing the task), cost (the probability that the cost for executing the task is less than expected cost), timeliness (probability that the task will be completed before the estimated task completion time) and quality (probability the quality of result is sufficient). Their approach enabled agents to select suitable teammates based on the four dimensions and depending on the situation.

Ahn et al. (2007, 2008) defined three dimensions for agents in their model, including reliability (probability for fulfilling the task requirements), quality (quality of the provided services) and availability (being available as a teammate). They also used multiple dimensions for tasks. In order for an agent to maximise its reward, it must consider the helpfulness value of other agents with respect to various dimensions before it selects a teammate. In addition, agents in their system were able to adapt in dynamic situations using reinforcement learning (Sutton and Barto, 1998) technique. Reinforcement learning enabled the agents to change their preferences towards selecting teammates and according to the requirements of the new circumstances.
3.2 Modelling agent’s behaviour in cooperative scenarios

Our framework uses CPNs to model agents’ IPs. CPNs are a distributed modelling technique and allow for modelling concurrent actions of a single agent. They also allow modelling concurrent actions of multiple agents. In addition, CPNs were used to model complex tasks. Task CPNs represent the order of subtasks and their interdependencies. The use of task CPNs facilitates agent management of the order in which various requirements must be performed.

This section introduces cooperation methods that have used PNs and FSMs for modelling agents’ actions and behaviours.

3.2.1 Cooperation methods based on Petri Nets

Chainbi et al. (1996) used CoOperative Objects, which is an OO language that supports concurrency through the use of PNs. The authors employed PNs to model behaviour and cooperative activities of agents in a Prey-Predators scenario. Their approach supports distributed situations; however, the behaviour of agents in their framework is static and cannot be altered or adapted at run-time.

Duvigneau et al. (2003) used reference-nets (Kummer et al., 1999) to develop a Concurrent Agent Platform Architecture (CAPA) that allows inter-platform communication among agents being hosted on the same machine or different hosts. They also used a simulation engine called Renew (Kummer et al., 1999) to execute their nets. Their platform supports hierarchical nesting in protocols by means of reference-nets, and allows for defining nets within nets. Duvigneau et al. (2003) employed aspects of higher-level PNs, such as channels, reference-nets, and synchronization that are somewhat different from (or extensions of) Coloured Petri Nets.

Costelha and Lima (2008) used PNs for modelling robots’ behaviours. In their model the execution of robot behaviour is simply the execution of the robotic tasks. They also introduced a supervisor to control the execution of behaviours and provided a formal representation of their approach for a robotic soccer scenario. They did not provide any implementation for their suggested approach.

Hansen et al. (2004) provided a methodology for coordination in agent-based manufacturing based on PNs. Their approach enabled the agents to use PNs for their interactions,
and allowed them to alter their plans and actions in order to improve their performance. Hansen et al. (2004) focused on developing mechanisms that facilitate cooperation among a team of agents while avoiding unsuitable situations. This was achieved by performing analysis on individual collaborative networks and converting them into acyclic PNs. They formally tested their approach using a manager and several bidders but they did not provide any implementation details.

Classical PNs are not suitable for modelling real-time systems due to their lack of support for modelling time (Montano et al., 2000). Most MAS frameworks do not consider the fact that actions are non-instantaneous when using physical agents. Actions are not certain in non-deterministic systems (for instance, a committed action may not be performed due to physical faults in the physical device). A few works have tried to address these issues using the PN formalism. Generally there are two methods for incorporating time into PN models. One method, known as timed Petri Nets, considers associating a delay with instantaneous actions (transition firing), while the other method associates a duration with transition firing. The first method is called timed Petri Net. Ramchandani (1974) used timed PNs, where they assigned a deterministic and finite duration with firing each transition. Their model does not consider the fact that the duration of the activities are not fixed in real time applications. Huang et al. (2009) used a Fuzzy timed PN for modelling multi-robot cooperation. However, their model employs autonomous agents (for definition refer to Section 1.2.1). It considers static time intervals for modelling timed activities of robots. The second method for modelling time is to assign a delay with some probabilistic distribution to each transition. This has resulted in stochastic and generalised stochastic, PNs. Such PNs have been used for parameter analysis and evaluation of the system performance. However, stochastic methods are not suitable for modelling real time since it is impractical to guarantee absolute properties for time (Montano et al., 2000). Our framework considers a robotic context, and since in robotic scenarios robots actions are not instantaneous, the timed PN was used to model agents non-instantaneous actions. The transitions are fired instantaneously, but there is a delay associated with the transition firing (such actions include moving and waiting to receive a response).

PNs have also been used for modelling coordination in robotic scenarios. Palamara et al. (2008) and Ziparo et al. (2010) introduced Petri Net Plans (PNPs), which is a language for
modelling robots behaviour based on PNs. It allows for modelling robots sensory behaviour in addition to modelling concurrent behaviour of agents. Since PNP's are based on PNs, they support formal analysis of robot plans. In addition, they enables the developers to model the behaviour of a single robot as well as modelling cooperative behaviour of a team of robots. PNP’s design supports distributed execution of multi-robot plans. Ziparo et al. (2010) used operators to combine different atomic actions of robots. The operators in PNP's are sequence, interrupt, fork and join. Sequences combine actions, and interrupts interrupt actions in case of failure. The fork operator is a way for creating multiple threads from a single thread of execution. Finally, the join operator combines multiple threads into one thread of execution. The authors tested their system for a foraging scenario (using three robots) and robotic soccer (using three robots). It is not clear how PNP's deal with scenarios that require tight and concurrent coordination among more than two robots (their systems only allows for coordination between two robots). In addition, no formal representation was used for representing tasks.

3.2.2 Cooperation method based on Finite State Machines

FSMs have been used for coordinating teams of robots. Risler and von Stryk (2008) used Extensible Agent Behaviour Specification Language (XABSL) for modelling Cooperative Multi-Robot Systems. XABSL is a modular and scalable tool that facilitates the engineering of complex behaviours in multi-agent systems. Their approach supports concurrent execution of multiple state machines by employing hierarchical finite state machines. In their architecture, the agents behaviour is decomposed into simple state machines called options. The composition of these options results in a complex hierarchical state machine. Each option consists of a set of actions that are executed while their states are active. Their model supports concurrent execution of multiple actions by allowing calls on multiple options.

Several frameworks have used FSMs to model agents behaviour including (Gat, 2002). The authors claim that it facilitates the development of behaviours in real time (Gat, 1992). Buch et al. (2010) used FSMs to model robots behaviours by using a playing-robot environment. The playing-robot is an environment that can be used to study the interaction between humans and robots. The authors used robotic agents that are equipped with sen-
sors and a sound-processing capability to entertain visitors. They used Lego Mindstorms NXT (Mindstorms, 2010) and Khepera III (Khepera, 2011) robots for their experiment. The robots were controlled remotely from different computers (the robots could not communicate directly). They used a special type of FSM that is called a *deterministic transducer*. The *deterministic* characteristic means that the robots can determine their next states using information of the current states, and the *transducer* means that the FSM can produce outputs. In their experiments the two robots (Mindstorms NXT and Khepera) had to change their movement patterns when a visitor came close to the Khepera robot. Their approach did not consider concurrency.

Morley and Myers (2004) used FSMs to represent tasks in the SPARK framework. Kube and Zhang (1996) employed FSMs for achieving cooperation among a team of robots for a box-pushing task. Their approach required the robots to decompose the task into a linear series of subtasks and then build a FSM controller that executes the subtasks. Marino et al. (2009) used FSMs for selecting appropriate actions for robots in a multi-robot patrolling scenario. In their work, the robots decide their next actions based on their sensing capabilities. The decision mechanism of the robots is based on FSMs, which play the role of a *supervisor* for each robot. The use of FSMs allow encoding all the possible transitions between actions. However, they are not suitable for situations consisting of large action domains, or situations considering highly dynamic environments.

### 3.3 Discussion

Here we briefly identify the limitations of the pre-existing work and discuss the advantages of our approach.

Task allocation mechanisms, including market based, and coalition formation approaches, have mainly focused on maximizing agents revenues. They are inefficient for coordination of complex tasks that require a tight degree of coordination, for example those requiring coordination of interdependent and concurrent tasks. In addition, centralised market-based mechanisms including “combinatorial auction” (Fujishima *et al*., 1999), “reverse auction” (Emiliani and Stec, 2001) and Vickrey-Clarke-Groves (Mas-Colell *et al*., 1995) are not suitable for coordinating agent activities since they are not scalable. Task allocation mecha-
isms have also addressed the problem of modelling task interdependencies, for instance, LA-DCOP uses an AND constraint to support concurrent execution of subtasks (Scerri et al., 2005). Some authors have used task trees to represent interdependencies between various subtasks (Gerkey and Mataric, 2002; Zlot and Stentz, 2006; Sariel et al., 2007; Vig and Adams, 2006; Landén et al., 2010). However, task trees are hard coded in these systems. In addition, task trees cannot be directly mapped to execution. Our framework used the CPN technique (which is an extension of PNs) for modelling agents’ concurrent actions as well as coordination of concurrent inter-agent activities. The framework includes complex tasks, where normally there are interdependencies between subtasks that specify the order in which various subtasks must be performed. We have also used CPNs to model complex tasks and interdependencies at a higher abstraction level. The higher-level task CPN supports managing interdependencies between different subtasks of a complex task. The use of CPNs as a modelling technique has a further advantage, since by employing the JFern CPN execution engine (JFern is described in Section 5.2.2), the CPN coordination model can be directly executed by real-time agents.

Network-based task allocation mechanisms generally consider a fixed topology for the connection among agents (Abdallah and Lesser (2005); Sander et al. (2002); Weerdt et al. (2007)). Such assumptions do not hold in robotic contexts, since robots constantly move from one location to another thus leaving and entering visibility ranges of other different robots.

Multi-criteria decision making approaches allows agents to consider multiple criteria when selecting teammates. It enables agents to select appropriate teammates based on multiple criteria, and according to the circumstances. For instance, the quality of service may be of a higher importance when cleaning chemicals from a contaminated area. However, when dealing with a fire that is spreading fast due to high wind, the speed of executing the task is far more important than hiring the best fire fighters. In addition, this approach suits dynamic situations where preferences of agents change dynamically according to the requirements of the new situation. In our framework, agents use a multi-criteria decision making approach for selecting teammates. The agents may change their values for different criteria according to the requirements of different situations. Agents in our framework use reinforcement learning mechanisms similar to Ahn et al. (2007, 2008) and Soh and Li (2004) to support
agents decision making in changing circumstances.

Cooperation mechanisms that employ modelling techniques such as PNs and FSMs enable the developers to visually represent coordination mechanisms. In addition, they support formal analysis and validation of their models. PNs are more suitable for our framework due to their expressive power for modelling concurrency. Thus we used Coloured Petri Nets for modelling agents’ concurrent activities. Duvigneau et al. (2003) used reference nets which are extensions of Petri Nets. Their platform supports concurrency by enabling each agent to have its own processing thread. However, they did not support modelling complex tasks. In addition, they did not test their system using physical robots. Ziparo et al. (2010) introduced PNPs which are an extension to PNs. Although they tested their system using physical robots, they did not test their systems for situations where tight coordination is required for executing complex and interdependent tasks. FSMs have also been used for modelling agents’ activities (Risler and von Stryk (2008); Kube and Zhang (1996)). However, their expressive power is not enough for modelling complex interactions, especially those with some degree of concurrency.

3.4 Summary

This chapter has provided a review of the literature with respect to cooperation in multi-agent and multi-robot systems. It surveyed task allocation mechanisms, including market-based, coalition formation and multi-criteria decision making. Market-based (or auction based) mechanisms focus on improving agents utility by enabling them to select the best offer (or bid). Coalition formation techniques focus on team formation. Such a mechanism enable agents to join a group if it leads to an improvement in the utility of the team or the overall system. We have separated task allocation mechanisms for robots from task allocation mechanisms for agents. This is due to the fact that assumptions made in some multi-agent system work are not practical in multi-robot systems. We also described the task modelling techniques used in a few of these works. Multi-criteria decision making is another field that has addressed task allocation. Such a mechanism consider multiple factors when selecting teammates.

FSMs and PNs have been used to model agent behaviour in cooperative activities. Among
all the modelling techniques introduced in Section 2.8, PN is the only technique that supports modelling of concurrency. As a consequence PNs have been used to model concurrent behaviours of agents in our framework. This is an important feature, since in realistic scenarios agents may need to perform several actions concurrently.
Chapter 4

Framework Description

Open distributed multi-agent systems are composed of multiple independent agents that perform dependent tasks. Typically, agents deployed in open environments may have different expertise and desires and act as self-interested entities to achieve their corresponding goals. In such systems agents may confront different types of domains and must be able to interact with other agents to build a team (Ahn et al., 2007). Another important feature of open multi-agent systems is their adaptability. Environmental conditions may change and therefore agents require a mechanism to identify the most influential aspects of multiple constraints. Such mechanisms enable the agents to alter their behaviour according to new circumstances.

We have designed a framework for facilitating cooperation among a team of robotic agents in situations where agents are required to form a team in order to complete complex tasks. The aim of the framework is to test the behaviour of the agents prior to their deployment on physical robots. The framework considers multiple dimensions for teammate selection based on attitudes and capabilities. All agents are self-interested entities and there is no central mechanism to control the system. The framework allows the agents to interact with their neighbouring agents and facilitates cooperation when required.

This chapter introduces the basic components of our simulation framework including agents and an environment model. It starts by building the ‘base’ framework that employs primitive agents. The framework is used for testing the behaviour of robotics agents that are required to form a team and move to a location under temporal constraints. It presents some
preliminary results that were achieved by using the simulation framework. The last part of this chapter introduces the ‘multi-threaded’ framework which employs autonomous agents. The multi-threaded framework improves the autonomy of agents and allows them to interact in a more realistic concurrent manner. It allows the agents to include the notion of time in a more realistic way, since all the agents perform their actions concurrently. We also demonstrate how the framework enables the agents to adapt according to various circumstances in order to improve their reward.

4.1 Problem domain

In this thesis a disaster application scenario is examined for illustration in which a group of people are trapped in an unsafe area, and a group of robots with different capabilities may be required to save the victims. In the disaster scenario domain, agents must search for victims and identify their locations. The victims may be buried under the rubble, and there may also be a fire in an area where the victims are found. In such situations, agents are required to extinguish the fire and clean up the rubble before they can rescue the victims. Once the fire is extinguished and the rubble is cleaned up, the victims must be carried to the nearest safe area. In our rescue example, certain capabilities for agents are required. Agents with fire fighting equipment are required to extinguish the fire, and agents with shovels are required to clean up the rubble. Finally, agents with carrying capabilities are required to carry the victims to the nearest safe location.

4.2 Simulation model

We simulated a physical environment by a grid where the area is divided into several spatial regions. The agents move from one location to another in order to find or perform a task. The simulation environment was established by defining a rectangular grid of cells, with each cell representing a distinct location. With each movement, an agent moves to one of its adjacent cells (agents move along to North, South, East and West). RFID (Radio Frequency Identification) tags are assumed to be deployed in each region and contain some information with respect to the geographical coordinates of the regions. In addition, each tag is assumed
to store the task information if there is any task in that region. It is assumed that the agents are equipped with limited range RFID readers that allow agents to position themselves in the environment by reading the coordinate information from environment tags. An agent that finds a task must then find teammates and perform the task together with them before the time expires. To achieve this, an agent must find appropriate teammates in a timely fashion. Each agent participating in performing a task is rewarded by a monitoring agent which oversees the simulation environment. The goal of each individual agent is to maximize its own reward.

### 4.2.1 Model of the environment

There are various tasks (e.g., victims requiring rescue) that are located at unknown locations in the environment. These tasks need different equipment depending on each situation. The framework includes several agents with various equipment (or capabilities) for the rescue operation. The environment is formally represented as

\[ \{A, TS, M\} \]

where

- \( A = \{ a_i, a_j, ..., a_n \} \) is the set of agents distributed randomly in the environment.
- \( TS = \{ ts_i, ts_j, ..., ts_n \} \) is the set of task descriptions.
- \( M \) is the monitoring agent which contains the criteria and policies for rewarding the agents.

### 4.2.2 Agent capabilities

Agents have different capabilities that are useful in satisfying different task requirements. Since we are considering a robotic scenario, we assume that the capabilities of agents are fixed and do not change over time. In the context of our disaster scenario, the capabilities can refer to various equipment required for rescue operation, such as shovels and fire extinguishing equipment. The shovel can be used to clear debris from the area, and the fire extinguishing equipment may be used to extinguish the fire. In addition, agents with tail lifts or ramps are required in order to facilitate loading and carrying victims to a safe place. The
A set of capabilities is represented as

\[ C = \{c_i, c_j, \ldots, c_n\} \]

where \( C \) is the set of capabilities of the agents. This chapter assumes that agents may have more than one capability. Each capability \( c_i \) is represented by a value and refers to the quality that the agent can provide for the \( i^{th} \) requirement of the task.

### 4.2.3 Task definition

Tasks (i.e., victims) are distributed randomly in the environment and have different requirements that must be satisfied by the different capabilities of the agents. A task is represented as a tuple:

\[ < R, t, w > \]

where

- \( R = \{r_i, r_j, \ldots, r_n\} \) is the set of requirements.
- \( t \) is the time constraint of the task.
- \( w \) is the basic reward that a team receives by performing the task. The reward is distributed equally among the participating agents if they can perform the task before time expires.

It is assumed that each \( r_i \) requirement of the task can be fulfilled by \( c_i \) capability of agents.

### 4.2.4 Agents attitude

In addition to capabilities, each agent has attitudes that impact its decision strategy over teammate selection. Social and behavioural psychologists describe attitude as a predictor of behaviour (Tesser and Shaffer, 1990). Therefore the concept of attitude can provide a mechanism to modify the agent’s decisions. Since the attitude can be learnt from agents’
interaction with other agents rather than being innate, it enables the agents to adapt to changing environments by identifying the most appropriate set of attitudes according to various circumstances. In this chapter each agent is assigned to have the following two attitudes:

- **att_nearness**: Attitude toward nearness; This refers to the agent’s inclination to seek teammates that are physically close to the task.

- **att_quality**: Attitude toward quality; This refers to the agent’s inclination to find teammates that provide high quality of service.

Each attitude in this study is normalised to range from 0 to 1. For example, when an agent has the set of attitudes \{0.2, 0.8\}, it indicates that the agent has a low attitude towards nearness and a high attitude towards quality. If an agent has a high attitude towards nearness, then it prefers teammates that have the least physical distance to the specified task. Similarly, if an agent has a high attitude toward quality, then it prefers agents which will provide the highest quality with respect to the task requirements.

### 4.2.5 Reward mechanism

There is no single method for rewarding agents that are involved in a cooperative task. The reward mechanism changes significantly depending on the scenario. For instance, in a cooperative task like a football game it may be reasonable to reward agents equally. However, in extreme tasks like oil-well fire control where any subtask of the fire control can be extremely critical for containing the fire, reward can vary greatly for various participants. At certain points of the task performance, an agent with great skill may be paid extremely well to perform a critical subtask in a timely fashion. Therefore for certain tasks where professionals are required, the reward of each participant changes according to the importance of each task. The following quote from Red Adair\(^1\) shows the value of hiring professionals from Red’s point of view.

“If you think it is expensive to hire a professional to do the job, wait until you hire an amateur.”

\(^1\)American fire fighter and the founder of Red Adair Co. Inc., His team pioneered fire-fighting techniques and tools. After the Gulf War in 1991 he participated in controlling the oil-well fires in Kuwait.
Various mechanisms have been proposed for rewarding agents in multi-agent literature. Wooldridge (2009) refers to two cooperative games in his book “An Introduction to Multi Agent Systems”. These mechanisms are: The core and Shapely value. The former focuses on evaluating the stability of a coalition, while the latter focuses on how the reward must be implemented. The core stability specifies the possibility for team formation, but it does not specify how the reward must be divided between agents in a team. The Shapely value suggests that agents in a team must be rewarded according to their contributions. This is a meritocratic approach that means agents who perform better get a higher reward. Although the Shapely value seems suitable for some situations, it does not suit our framework. In our framework all the agents are rewarded equally to encourage teamwork. The agents are rewarded if the task is performed within a specified time constraint.

Each task requires agents with certain capabilities and has an associated reward that is proportional to the task size. In our work, we assume that an agent can perform a fraction of a task according to its capabilities. A team of agents can perform some portion of a task if they cannot satisfy all the task requirements. When agent $a_i$ works on a task $t$ with a number of agents as a team, and the team completes the task, then each agent participating in completing the task receives the following reward:

$$
\sum_{i=1}^{n} \min\left(\frac{\sum_{j=1}^{m} (a_j)_{c_i}}{r_i}, 1\right) \times \frac{w}{m}
$$

(4.1)

where

- $n$ is the number of task requirements.
- $m$ is the number of participating agents.
- $r_i$ is the $i^{th}$ requirement of the task.
- $(a_j)_{c_i}$ is the value of the $c_i$ capability of the agent $a_j$ for $i^{th}$ requirement.
- $w$ is the basic task reward.
- $\min$ returns the value associated with the ratio (capability/requirement) if it is less than 1, otherwise it returns 1.

For example assume that two agents perform a rescue operation which offers the basic reward of 5. Then the amount of reward that each agent receives is 2.5. Accordingly, if two
agents just finish 80% of that task, then each agent receives a reward of 2.

4.3 Agent strategies

An agent strategy is determined according to its attitudes. In our system the capabilities of agents and their distance from tasks have an impact on the agents’ performance. Therefore the team-selection strategies are designed so they reflect the effect of distance and agent capabilities. Agent strategies also allow measuring the performance of the system in scenarios when time is critical and agents must perform their tasks within a specified time.

Table 4.1: Various agent strategies, based on attitudes. An agent strategy is determined based on its attitudes.

<table>
<thead>
<tr>
<th>attitudes (nearness, quality)</th>
<th>strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, high)</td>
<td>Best possible</td>
</tr>
<tr>
<td>(low, high)</td>
<td>Best available</td>
</tr>
<tr>
<td>(high, low)</td>
<td>Nearest available</td>
</tr>
</tbody>
</table>

We designed some specific team selection strategies and performed simulation experiments. The code defining the agent strategies has been written in a separate module so that it can be easily replaced by a different module representing different strategies. Some strategies require the agents to complete the tasks in their entirety (Best possible), while other strategies allow partial performance of the tasks (Best available, Nearest available). Table 4.1 shows the various strategies.

4.3.1 Nearest available strategy

Agents that have a high attitude towards nearness and a low attitude towards quality employ the Nearest available strategy. An agent that uses this strategy selects agents that lie at the least distance to the task. Agents with this strategy receive partial rewards in proportion to the completed part of the task. Algorithm 4.1 shows the pseudocode for calculating the agents distance from the task location. The distance is calculated based on the fact that
agents can only move in one of the four cardinal directions (N,S,E,W). Algorithm 4.2 shows the pseudo-code for selecting the nearest agents for *Nearest available* strategy.

**Algorithm 4.1:** Pseudocode for calculating an agent’s distance from a task.

**Input:** task location(TL) and agent location (AL)

**Output:** the agent distance from the task

```plaintext
/* retrieve the agent’s coordinate */
1 AL_x = retrieve x coordinate from AL
2 AL_y = retrieve y coordinate from AL

/* retrieve the task coordinate */
3 TL_x = retrieve x coordinate from TL
4 TL_y = retrieve y coordinate from TL

/* calculate Manhattan distance */
5 distance = \(|AL_x - TL_x| + |AL_y - TL_y|\)
```
4.3.2 *Best available strategy*

Agents that have a high attitude towards quality and a very low attitude towards nearness employ the *Best available* teammate strategy. These agents may select teammates that have only some of the expertise required by the task, and they receive a partial reward for the part of the task that is completed. When an agent $a_n$ looks for hiring teammates, the quality of each available agent in the neighbourhood ($a_j$) is calculated based on the following equation:

$$Q_{a_j} = \sum_{i=1}^{n} r_i \times (a_j)_{c_i}$$

(4.2)

where

- $Q_{a_j}$ is the quality of capabilities of agent $a_j$.
- $n$ is the number of task requirements.
- $(a_j)_{c_i}$ is the value of the $c_i$ capability of the agent $a_j$ for the $i^{th}$ requirement of the task.
- $r_i$ is the $i^{th}$ requirement of the task.

The $c_i$ capability fulfils the $r_i$ requirement of the task. The capabilities of each agent for each task requirement are paired with that requirement. For example, the total of fire extinguishing capabilities of the agents is paired with the fire extinguishing requirement of the task. So if for example the agents need to work in area with a large fire, then this experiment weighs the fire extinguishing capability of an agent more heavily. For teammate selection each agent considers all available local agents and ranks them based on their helpfulness value. Given the quality of capabilities, the agent calculates the helpfulness of each possible teammate according to the following equation:

$$H_{a_j} = (Q_{a_j} \times a_{natt-quality}) + (a_{natt-nearness} \times dist_{a_j})$$

(4.3)

Where

- $H_{a_j}$ is the helpfulness value of agent $a_j$ as a teammate.
- $a_{natt-quality}$ is the attitude of the agent $a_n$ towards quality of teammates.
- $a_{natt-nearness}$ is the attitude of the agent $a_n$ towards distance.
- $dist_{a_j}$ is the inverse of the agent distant to the task.
Algorithm 4.2: Pseudocode for Nearest available strategy.

Input: available agents (AGs) and task (T)

Output: teammates that lie at the least distance to the task (T)

/* retrieve the subtasks and put them into Ts */
1 foreach subtask (s) in task (T) do
2    add s to Ts
3 end

/* calculate the distance from task (T) for all the available agents (AGs) */
4 foreach agent (A) in AGs do
5    if A has capability to satisfy a subtask (s) of Ts then
6        distance = calculate A’s distance from task using Algorithm 4.1
7        Add distance to DL
8    end
9 end

/* sort DL ascending based on their distance from the task */
10 sort DL

/* Q stores the list of found capabilities required by the task */
11 foreach agent (A) in DL do
12    foreach subtask (s) in Ts do
13        if required capabilities are not found in Q, and A has the capability to satisfy s then
14            add A to teammates
15            add A’s capabilities to Q;
16        end
17    end
18 end
In this equation an agent’s attitude towards quality is associated with the quality that the agent can provide with respect to task requirements, and an agent’s attitude towards nearness is associated with the distance of the potential teammate from the tasks. According to this equation, an agent that uses this strategy favours quality of work more than distance. Agents with this strategy rank all the available agents based on their helpfulness values (equation 4.3) and then select their teammates. If there are several agents with the same helpfulness value then the agent that lies at the least distance to the task is selected.

Algorithm 4.3 shows the pseudo-code for selecting teammates for an agent that has the Best available strategy. Initially, all the subtasks are identified and the agent calculates the helpfulness value for all the available agents. Then it sorts the agents based on their helpfulness values and selects its teammates.

4.3.3 Best Possible teammate strategy

Certain tasks must be performed continuously and completely. For instance, in situations where explosive materials exist near an area that is on fire and contains elderly victims trapped under the rubble (for instance situations where an earthquake causes a retirement home to collapse). For such tasks the fire must be extinguished completely to avoid the risk of a blast. In addition, the elderly patients must be rescued quickly as most of them have medical conditions. Therefore there is a need for strategies that disallow partial performance for tasks. An agent with this strategy only recruits other agents as its teammates if they can complete the task. Agents that have a very high attitude towards quality employ the Best possible teammate strategy for selection of teammates. It is formally represented as:

$$\forall r_i \in R \implies \sum_{j=1}^{m} (a_j)_{c_i} > r_i$$  \hspace{1cm} (4.4)

where

- $m$ is the number of teammates.
- $r_i$ is the $i^{th}$ requirement of the task.
- $(a_j)_{c_i}$ is the value of the $c_i$ capability of agent $a_j$ that corresponds to the $i^{th}$ requirement of the task.
Algorithm 4.3: Pseudocode for Best available strategy.

**Input:** available agents (AGs) and task (T)

**Output:** teammates with best quality for performing task (T)

/* retrieve the subtasks and put them into Ts */

1. foreach subtask (s) in task (T) do
   2. add s to Ts
   3. end

4. while AGs is not null and Ts has a subtask s that is not satisfied do
   5. foreach agent (A) in AGs do
      6. if A has capability to satisfy Ts then
         7. H = calculate A's helpfulness value
         8. add A and H to AQs
         9. end
      10. end
   11. /* sort agents (AQs) descending based on their helpfulness value */
   12. sort AQs based on H and inverse value of distance
   13. add the first agent (A) from the top of the list (AQs) to teammates
   14. deduct from Ts the requirements fulfilled by A's capabilities
   15. remove A from AGs
   16. end
4.4 Adaptation

Our framework is an open agent system in which agents can join and leave at any time. In addition, the circumstances that agents perform under may change due to various reasons. Therefore the agents must be enabled with adaptive mechanisms that enable them to adapt to new conditions in order to maximize their rewards. In the work presented in this section agents can adapt to new conditions by changing their attitudes. Agents change their attitudes based on the feedback that they receive from the environment. For instance, there might be a situation when agents receive low rewards because required task completion time is very restricted. In this situation, agents should learn to decrease their attitude toward quality and increase their attitude toward nearness. Subsequently agents will come to favour the nearness of teammates to tasks over the quality that they can provide. A reinforcement learning mechanism adopted from (Sutton and Barto, 1998) is used to alter the agents attitudes. An agent performs with its current set of attitudes for a certain period of time and stores a copy of the attitudes and the total reward received. Then the agent increases its attitude towards one dimension and starts performing. Because attitude has two dimensions (nearness and quality), increasing one dimension is associated with decreasing the other dimension. Then the agent performs for the same amount of time with its new set of attitudes. If the reward that the agent receives is more than the reward gained by its previous set of attitudes, it continues to change its attitudes along the same lines; otherwise, it increases its attitude towards the other dimension. The following reinforcement learning formula is used to update the agents attitudes:

\[ \alpha_{t+1} = \alpha_t + \beta(Q(\alpha_{t+1}) - Q(\alpha_t)) \]  

(4.5)

where

- \( \alpha_{t+1} \) is the new value for attitude of the agent.
- \( \alpha_t \) is the previous value for the attitude of the agent.
- \( Q(\alpha_{t+1}) \) is the actual reward received for new attitudes.
- \( Q(\alpha_t) \) is the actual reward received by previous attitudes.
- \( \beta \) is the learning rate.
4.5 Experiments

After formulating this model, a series of experiments was conducted to study the effect of agent attitudes on agent performance. In these experiments the agents were required to identify tasks and form teams. The tasks have a time constraint, and a team must perform a task before the time expires. The experiments assume that a team is made up of two agents although it can be easily changed to include more agents (none of the other chapters of this thesis have any restriction on the number of agents in a team). The simulation environment is a grid of 100 by 100 cells. Each cell is assumed to be equipped with an RFID tag that contains some information with regards to the coordinates of the cell and the presence of tasks (e.g., trapped victims). If there is a task in a cell, then the RFID tag also has the task information. There are 100 agents with different capabilities. The capabilities can be for example shovels, fire extinguishing and victim carrying equipment. An agent may have any one of or a combination of these capabilities. Agents can only move one cell at a time in one of the four cardinal directions (N,S,E,W).

The capability values indicating quality level of expertise (skill) were normalized to range from 0 to 1 and were selected randomly. All task requirements were also set randomly and range from 0 to 1. These assumptions were made to maintain the generality of the framework so it could be applied to different domains. These selected ranges can simply be replaced and set to any other user-defined values. Since our agents at this point are primitive agents, an iterated round-robin approach was used to execute the agents.
Figure 4.1: Example of various agent visibility ranges. The dark grey area
defines the task visibility range for the agent $a_1$, and the light grey area
together with the dark grey area shows the visibility range of agent $a_1$ for
other agents. $T_1$ and $T_2$ represents locations that have victims without and
with fire respectively.

Figure 4.1 shows a schematic diagram of part of the experimental grid, where the visi-
bility range for the task is defined as 2 cells. The area with the dark grey shade shows the
task visibility range for agent $a_1$. The area with the light grey shade shows the visibility
range of agent $a_1$ for detecting other agents. This area also includes the area with the dark
grey shade. In this example, the visibility range for detecting other agents is 5. The visibil-
ity range for detecting other agents was higher than the visibility range for detecting tasks
(victims). This reflects common practical situations, as sensors that are used to locate vic-
tims in disaster situations usually have a short range. On the other hand, physical robots are
likely to employ greater-ranged WiFi or Bluetooth technology for inter-agent communica-
tions. In the experiment, there are 1000 tasks with different requirements placed randomly
in the environment.
Table 4.2 shows the parameters used in the simulation. The visibility range for all agents except agents with *Nearest available* strategy, is set to 10. Since the grid size is relatively large, the visibility range was set to a high number so the agents can see several other agents in their visibility range. This gives agents a choice when selecting teammates thus the impact of different strategies can be observed. The visibility range of agents with *Nearest available* strategy (6) is slightly shorter than the visibility range of the other agents. This reflects the agents intention for selecting closer teammates. The visibility range for tasks is set to 5. This number was selected based on average distribution of tasks on the grid. The reward is an arbitrary number (5), and is the same across different versions of the framework described later in this thesis.

<table>
<thead>
<tr>
<th>Table 4.2: System parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter</td>
</tr>
<tr>
<td>Number of agents</td>
</tr>
<tr>
<td>Number of tasks</td>
</tr>
<tr>
<td>Visibility range for agents with <em>Nearest available</em> strategy</td>
</tr>
<tr>
<td>Visibility range for other agents</td>
</tr>
<tr>
<td>Visibility range for tasks</td>
</tr>
<tr>
<td>Basic reward</td>
</tr>
</tbody>
</table>

A time step refers to a simulation cycle in which agents observe their environment, select a teammate (if possible), move one cell and possibly work on a task. If an agent has selected a task, then it moves one cell toward the task; otherwise it moves randomly to one of its neighbouring cells.

In order to account for the possible variations in the experiments, I chose to repeat each experiment 10 times. Since it was observed that the standard deviation over ten runs was only 0.01, I concluded that there was no need to consider a larger number of execution.
4.5.1 The effect of task time on agents reward where the agents employ various teammate selection strategies

The aim of the first experiment is to examine the effect of various task time constraints on the total reward achieved by agents in the system. In this experiment there are three groups of agents in which each group employs one strategy (Best available, Best possible and Nearest available strategy). The agents work on the tasks until all the tasks are completed or their time expires. The total reward achieved by each group is measured under various task time constraints of 5, 10, 20 and 30 time steps. These parameters were selected experimentally in order to show the effect of task time constraints on agent performance.

Figure 4.2 shows the total reward achieved by each group when employing different strategies under different task time constraints. It shows that when task time is very limited (time constraint=5), agents with the Nearest available strategy outperform the other groups. This is the effect of selecting the closest neighbour as a teammate. Moreover, the low reward of the Best available and Best possible teammate strategies are mainly due to selecting teammates with respect to quality of service that may be further away from tasks. Therefore there may not be enough time for teammates to reach the task before time expires.

When the time constraint is more relaxed, the Best available strategy starts outperforming
the other strategies as the result of selecting higher quality teammates. It also shows that agents with the *Best possible* strategy perform worse than other groups under various time constraints. This is caused by the perfectionist attitude of these agents. An agent with the *Best possible* strategy that is looking for a teammate does not select teammates unless the agents as a team are capable of accomplishing the tasks in their entirety. Since there may not be agents with complementary capabilities in the neighbourhood, the agent may have to drop its current task and find another task to perform. Therefore these agents spend most of their time looking for teammates and not performing a task.

Figure 4.3 depicts that under various time constraints the agents that employ the *Best possible* strategy complete more tasks fully (as opposed to partial completion of the tasks) than other agents. It also shows that as the time starts to relax, agents with the *Best possible* strategy manage to complete more tasks which is due to having enough time for completing tasks.

### 4.5.2 The effect of learning and adaptation on agent reward

The aim of this experiment is to show how adaptation can improve the performance of the agents. This experiment could be conducted in various ways, but we chose the following scenario in order to show the impact of the adaptation in the framework. In this experiment
agents with the *Best available* strategy were employed, and the time constraint of tasks was set to a low value of 5 units. Two runs of the simulation were performed. In the first run agents did not learn, and in the second run agents changed their behaviour based on their past performance (if required).

Figure 4.4 shows the total reward gained by agents when they learn and do not learn. Since the specified task time was very restricted, agents eventually changed their attitudes by using the learning mechanism described in Section 4.4; they increased their attitude towards nearness and decreased their attitude towards quality. By changing these attitudes, agents eventually adapted their decision strategies to the current condition and changed their strategy to the *Nearest available* strategy.

![Figure 4.4](image)

**Figure 4.4**: The impact of learning on agent reward. In one experiment the agents were not learning and in the other experiment the agents were learning by using the mechanism defined in Section 4.4. The average reward is the reward of all the agents in the system in learning and non-learning scenarios averaged over 10 runs.

### 4.6 Multi-threaded agent framework

Our multi-agent framework was implemented by employing *primitive agents*. In order to create a more realistic simulation environment, the *primitive agents* were replaced by *autonomous agents*. *Autonomous agents* have their own thread of execution. The 'multi-
threaded' framework allows for a more realistic simulation environment by enabling agents to act in their own thread, where there is not a fixed time associated with different agents’ actions. The ‘base’ framework assumed that in each simulation cycle (or time step) each agent observes its surroundings, forms a team (if possible), moves one cell (if required) and possibly works on a task. Such an assumption is unrealistic as each of the processes may take a different period of time in realistic situations. The ‘multi-threaded’ framework enables us to capture the behaviour of agents in a more realistic concurrent environment. For instance, if an agent has a long queue of requests then the agent may be slow in responding to its requests. This is the case in real situations where busy agents may be slow in responding to their requesting agents.

Each agent in the framework makes use of java.lang.Thread so that each agent now has its own thread of execution. The multi-threaded approach allows the agents to execute and act concurrently. This feature is important, since the agents reside in an environment where temporal and spatial constraints are present and have an impact on agent performance.

4.6.1 Concurrent issues in multi-threaded framework

The agents in the multi-threaded framework share the tasks and environment, thus creating a need for a mechanism to deal with concurrent issues. The main issues are coordinating synchronisation and avoidance of deadlocks (Lea, 2000). Synchronisation can be implemented by using synchronised methods or synchronised statements. Synchronised methods provide a simple mechanism for preventing thread interference and memory consistency errors. If an object is accessible from multiple threads, then all changes to that object must be done through the synchronised methods. However, this method of synchronisation is not efficient, since it blocks a large portion of the code from executing, which reduces the concurrency and slows down the execution of the system. A solution to this issue is using synchronised statements. Synchronised statements block a statement that manipulates the object and specifies the object that provides the lock. This method of synchronisation allows for more concurrency by using fine-grained synchronisation. For example, in our framework all the methods of the agents that make changes to the task objects block the task object while an agent makes changes to it. Once the agent finishes with the task, then it releases the lock to that object.
Starvation and livelock are other problems that may occur in a multi-threaded environment. Starvation refers to a situation where a thread cannot access shared resources and therefore is unable to progress. For example, assume that an object provides a synchronised method that takes too long to execute. The regular invocation of such methods means that other threads that may access the same object frequently will often be blocked. Livelock refers to a situation where the action of one thread depends on the action of another thread (Lea, 2000). In such conditions, the agents become too busy responding to each other so they cannot resume work. Our multi-threaded framework considers all the above issues so the agents can execute and perform their tasks smoothly. This was mainly achieved by providing fine-grained synchronisation (blocking shared objects in small statements) and envisioning the interaction among agents.

4.6.2 Multi-threaded framework description

In the multi-threaded framework agents have two queues for holding agent messages: a request queue and a response queue. The former holds all the incoming messages to the agent, while the latter contains all the outgoing messages that are the agent’s responses to various requests.

Algorithm 4.4 shows the pseudocode which each agent runs constantly. In each execution cycle, each agent observes its environment and stores the tasks and other agents’ location information in its memory (observation). If an agent already has a task (that task could either have been found by the agent itself or could have been assigned to the agent by accepting a request from another agent), then it is considered to be engaged. If the agent is not engaged, then it selects a task in its visibility range and changes the task status to unavailable for the sake of the other agents’ information (the task object is blocked and then its status is changed to unavailable). In cases where there are several tasks, the agent selects a task that has the least distance to the agent. After finding a task, the agent starts looking for teammates. In order to find a teammate, the agent sends a request message to all available agents in its neighbourhood and waits for a certain amount of time to receive their responses.
Algorithm 4.4: Pseudocode for multi-threaded agents

Input: Request queue (RQ), response queue (RS) and task (T)

1. observe visibility range; // AGs=observed agents, Ts=observed tasks
2. if task (T) is null then
3.    find task T in Ts
4. end
5. if T is not null and agent is not part of a team then
6.    send request to AGs
7. end
8. wait for t period; // t is the waiting time period
9. repeat
10.   select the best response from RS and select the best request from RQ
11.   if $H$(selected response) > $H$(selected request) then
12.      form a team with the selected agent from the response queue
13.   end
14.   else
15.      respond to the selected request and remove the selected request from RQ
16.      wait for t period
17.   end
18. until agent is not part of a team and task time is not expired;
19. if T is not null then
20.   if task time has expired then
21.      drop the task
22.   end
23.   else
24.      move to task location
25.   end
26. end
27. else
28.   move randomly
29. end
When the waiting time is over, then the agent starts processing the received responses and requests. The agent ranks its response queue and request queue based on the helpfulness value of the agents. The helpfulness value is calculated using the helpfulness function defined in equation 4.3. If there is a request with a higher helpfulness value than all the responses, then the agent sends a response to the requesting agent. If the requesting agent does not select this agent within a designated time, then the agent repeats this process (corresponding to the loop on lines 9-18 of Algorithm 4.4) until it finds a teammate or the task time expires.

In order to ensure the CPU provides a fair execution time for all the threads, each thread was set with a high priority. In addition, at the end of each simulation run, a thread had to sleep for 20 milliseconds. This puts the current thread to sleep and allows the CPU to run the next thread. Since all the threads have a high priority, the CPU keeps switching between all the threads. Such a mechanism provides a relatively fair execution time for various threads, and gives an equal chance of execution to various agents. This is especially important when the time constraints of tasks are very tight and the agents’ responses should be given quickly.

4.6.3 Impatient strategy

The concurrent version of the framework has additional operational requirements and offers new features. The concurrent execution of agents allowed us to measure the impact of the temporal constraint in a more realistic fashion. A new attitude for agents was defined that considered the effect of response time. In our multi-threaded framework, once an agent sends a request, then it must wait for a certain period of time until it receives its responses back. Therefore we added the attitude towards response time which specifies how fast the agent needs its response.

- att−response time: Attitude toward response time. This refers to the agent’s inclination to find a teammate as quickly as possible.

Table 4.3 shows the attitudes of the agents in the multi-threaded framework. The new attitude, att−response time, allows for definition of the Impatient strategy. Agents with the Impatient strategy wait for a short period of time to receive their responses and then select their teammates.
Table 4.3: Agents strategies in the multi-threaded framework. An agent strategy is determined based on its attitudes.

<table>
<thead>
<tr>
<th>attitudes (nearness, quality, time)</th>
<th>strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, high, 0)</td>
<td>Best possible</td>
</tr>
<tr>
<td>(low, high, 0)</td>
<td>Best available</td>
</tr>
<tr>
<td>(high, 0, 0)</td>
<td>Nearest available</td>
</tr>
<tr>
<td>(0, 0, high)</td>
<td>Impatient</td>
</tr>
</tbody>
</table>

4.7 Multi-threaded framework experiments

A series of experiments using the parameters given in table 4.4, was conducted to study the performance of the multi-threaded framework. The simulation environment is a grid of 100 by 100 cells. There are 100 agents with different capabilities. There are 1000 tasks with different requirements placed randomly in the environment. The agents’ capabilities and task requirements are generated randomly at run-time. The system time was used for this experiment, and all the times are in milliseconds. A team is made up of two agents. There is a time unit associated with agents moving and performing tasks. The time unit must be long enough so the impact of the agents travel distance and task execution can be seen. The agent’s waiting time refers to the time that a requesting agent may wait until it receives a response from the requested agents.

4.7.1 The effect of task time on agent reward

In this experiment there are four groups of agents corresponding to each of the four strategies listed in Table 4.1. Four different runs of the simulation were run under various time constraints, where in each run, agents with one strategy were employed. The total reward achieved by each group was measured under various task time constraints of 500, 1000, 2000, and 3000 milliseconds.
Table 4.4: Simulation parameters for the multi-threaded framework. All time values are given in milliseconds.

<table>
<thead>
<tr>
<th>parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of agents</td>
<td>100</td>
</tr>
<tr>
<td>Number of tasks</td>
<td>1000</td>
</tr>
<tr>
<td>Visibility range for agents with Nearest available strategy</td>
<td>6</td>
</tr>
<tr>
<td>Visibility range for other agents</td>
<td>10</td>
</tr>
<tr>
<td>Visibility range for tasks</td>
<td>5</td>
</tr>
<tr>
<td>Waiting time for Impatient strategy</td>
<td>100</td>
</tr>
<tr>
<td>Waiting time for other strategies</td>
<td>200</td>
</tr>
<tr>
<td>Time unit</td>
<td>50 ms</td>
</tr>
<tr>
<td>Basic reward</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 4.5: The effect of agents’ strategies on agents reward under various task time constraints in the multi-threaded framework. The average reward is the reward of the individual agents that follow the same strategy averaged over 10 runs.

Figure 4.5 depicts that when time is very constrained (500 milliseconds), agents which employ the Impatient strategy outperform the other groups. Agents with the Impatient strategy do not wait to receive responses from all the requested agents and wait for a short period.
of time. Therefore they have a better chance of reaching the task within the specified time constraint. When time is less constrained (1000 milliseconds), then the Nearest available strategy outperforms the other strategies. These agents select the closest available teammates. Therefore the teammates may have enough time to move towards their task and work on it. However, other groups of agents may select a teammate that might be further away and therefore the team may fail to reach the task position before the time expires.

When time constraints are more relaxed (2000 and 3000 milliseconds) the Best available strategy outperforms the other strategies. This is the result of selecting high-quality teammates. The agents with the Best possible strategy perform worst under all the various time constraints. This is the effect of the perfectionist attitudes of these agents. This approach is useful when there is more incentive for completing a job.

4.7.2 The effect of referral on performance of agents with Best possible strategy

As was mentioned earlier, certain types of tasks may be required to be performed in their entirety. In such situations there is a need for mechanisms that enable the agents to find all their required equipment to perform the tasks. In our framework this situation may be advantageous for agents employing the Best possible strategy. An agent with the Best possible strategy does not select a teammate unless the collected agents as a team could complete the task. An agent with this strategy may simply fail to achieve some reward due to not having agents with complementary capabilities within its visibility field. In this case, a referral mechanism could be employed to improve the performance of agents with the Best possible strategy.

In multi-agent systems referrals involve systems whose members may follow a cooperative protocol by providing referrals to another agent(s). Referrals provide a means for sharing knowledge among agents, thus improving the allocation of resources. In this work, when an agent receives a request to perform a task where the agent itself does not have the required capabilities for performing the task and task time has not expired, then the agent checks its visibility range and if it can see any agent with the required capabilities, it sends a referral back to the requesting agent. An agent refers to another agent if the agent itself does
not have the required capabilities and task time has not expired.

In order to see the effect of referrals, three simulation runs were performed using agents with the Best possible strategy, and the task time constraint was set to be very relaxed. In the first run no referral was employed (number of hops=0). In the second run if the requested agent did not have the required capabilities, but it could see an available agent with the required capabilities, then it sent a referral to the requesting agent (number of hops=1). In the third run, the requested agent may ask its neighbours whether they could see an agent with the required capabilities (number of hops=2). The time constraint was set to be very relaxed, so agents had enough time to send and process referrals. Figure 4.6 demonstrates that when agents employ referrals (number of hops=1 and number of hops=2) their task completion performances improve.

![Figure 4.6: The effect of referral on agents performance.](image)

Although referrals improved the task completion performance of our system, they take time and therefore are not suitable for emergency situations where agents may need to quickly form a team and perform a task. Referrals are mainly suitable in scenarios where task time constraints are more relaxed, so the agents have enough time to communicate and find the required capabilities.
4.7.3 The effect of learning on agents reward

The aim of this experiment is to show how our agents are enabled with a learning mechanism that allows them to identify the best strategy in various situations. This mechanism can be used in dynamic situations so agents alter their strategies and adopt a strategy that maximises their reward. In this experiment agents with the Best available strategy were employed, and the time constraint of tasks was set to a low value of 500 milliseconds. Two runs of the simulation were performed. In the first run agents did not learn, and in the second run learning was employed.

The learning mechanism was similar to the one introduced in Section 4.4. An agent increases its attitude towards one dimension and performs for a certain period of time. If the reward that the agent receives is more than the reward that the agent achieved with its previous set of attitudes, then the agent continues to change its attitude the same way; otherwise, it increases its preferential attitude towards another dimension. Since time was very constrained, agents with the Best available strategy eventually increased their attitude towards time. By changing their attitudes, agents eventually adopted the Impatient strategy. Figure 4.7 compares the reward of the two experiments. It shows that the learning mechanism improved the reward of agents. In addition, since the Impatient agents have a relatively short waiting time for receiving their responses, the total time required to complete the tasks decreased.
4.8 Summary

This chapter introduced the basic components of our simulation framework that enables the coordination and management of a team of agents that operate in a distributed and spatial environment. The framework was designed to facilitate cooperation among agents in scenarios where agents are required to form a team in order to perform a complex task that cannot be performed by a single agent. In this framework, agents represent robots that move from one location to another. The agents have a limited visibility range and can only communicate with their neighbouring agents. The agents in this framework have various capabilities and attitudes, where the capabilities of the agents refer to the equipment that they carry which is different among agents, and the attitudes refer to agent’s desires for selecting teammates. Different values for different dimensions in agents’ attitudes leads to various teammate selection strategies, which impact the agents teammate selection strategies impact the agents’ reward.

Different strategies are suitable for difference situations. In this framework agents are capable of identifying the best strategy according to various circumstances. Agents alter their strategies by exploring various values for their attitudes. In addition, a referral mecha-
nism was employed to improve the performance of the agents with the *Best possible* strategy. Such agents do not perform well in connection with tasks where rewards are given to partial completions of tasks in comparison to agents with other strategies under the examined circumstances. This was due to the perfectionist attitude of these agents, since they perform a task with the aim of fully completing it (satisfy all its requirements). The suggested referral mechanism improved the chances of such agents to find the required capabilities. This can be important in circumstances where the full completion of a task is essential. The referral enables an agent to recruit agents with the required capability from other regions that is not visible to the agent itself.

The initial implementation of the framework employed *primitive agents* that did not exhibit autonomous and proactive behaviours. However, the second part of this chapter described how the *primitive agents* were replaced by *autonomous agents*. *Autonomous agents* have their own thread of execution. The multi-threaded framework created a more realistic simulation environment by enabling the agents to execute concurrently and exhibit autonomous behaviour. Our multi-threaded simulation environment comes closer to reproducing the concurrency conditions of real distributed multi-agent robotic systems.
Chapter 5

Modelling Agent’s Conversation

In the previous chapter we introduced the basic components of our cooperation framework. The ‘base framework’ employed *primitive agents* and used an iterated round-robin approach to execute the agents. However, that approach had two main issues concerning the autonomy of agents. First, agents did not have their own thread of execution and were not able to exhibit proactive behaviours. Second, each agent was able to directly invoke methods on another agent. This is against the principle of true autonomy in agent systems. In order to address those issues, the *primitive agents* were replaced by *autonomous agents* in the ‘multi-threaded’ framework.

This chapter explains how *autonomous agents* were replaced by *advanced autonomous agents* that are capable of exhibiting autonomous and flexible behaviours. We refer to this version the framework as ‘advanced’ framework. The ‘advanced’ framework uses the Otago agent platform (Opal) (Purvis *et al.*, 2002). Opal is a fully distributed agent platform that supports asynchronous communication among agents. The employment of the Opal agent platform facilitates the development of practical agent-based mechanisms that can be used to facilitate the formation of multi-agent teams in the context of spatial and temporal constraints. In order to help ensure that our coordination schemes and mechanisms can be deployed in complex and concurrent situations, we have developed a software approach that facilitates visual modelling and testing of the coordination system prior to its deployment in robotic systems. Our software approach employs Coloured Petri Nets (CPNs) (Jensen, 1997a) as a formalism to model agents’ conversations. CPNs are used to model multiple,
concurrent conversations in which a given agent may be engaged. The use of Opal agents
and CPNs ensures reliable deployment of our coordination schemes and mechanisms in real
situations where multiple agents interact and act concurrently.

This chapter introduces Coloured Petri Nets and their advantages for modelling agents’
conversations. It also introduces the Opal agent platform, and shows how Opal employs
Coloured Petri Nets to model agents’ conversations. We demonstrate our approach by ex-
amining some example scenarios and discuss the performance of the system under certain
conditions.

5.1 Modelling agent

In our framework, each agent interacts with other agents concurrently in order to find re-
sources required for performing complex tasks. Thus the interaction among agents in the
framework may be complex and makes decision-making a resource-intensive task. Therefore
there is a need for mechanisms that support modelling complex and concurrent interactions
among agents. Such mechanisms must enable the agents to keep track of their conversations
with various agents and provide appropriate responses to the requests of different agents.

Various mechanisms described in Section 2.8 can be used to model the agents’ Interaction
Protocols (IPs). In order for a modelling technique to suit our concurrent framework, it must
support concurrency. Among the defined modelling techniques in Section 2.8, Petri Net is a
well established formalism that supports concurrency. The next section elaborates on Petri
Nets and their extension, Coloured Petri Nets.

5.1.1 Coloured Petri Nets

A Coloured Petri Net (CPN) is an extension of a PN (PNs were introduced in Section 2.8.4). It
diffs from a PN significantly because its tokens are not simply blank markers, but have
data associated with them. A token’s colour is a schema or type specification. CPNs are
specified by a formal and graphical modelling language that was formulated by Jensen
(1997a,b,c). CPNs are useful for specifying, designing, and analysing concurrent systems.
In contrast to ordinary PNs, CPNs provide a very compact way of modelling complex sys-
tems, which makes the CPN formalism a powerful language for modelling and analysing complex systems. This is achieved by combining the strengths of PNs with the expressive power of high-level programming languages. PNs provide the constructs for describing the synchronisation of concurrent processes, and the programming language element provides the constructs for specifying and manipulating data values. Practical use of CPNs has been facilitated by the development of various tools, like CPN Tools (CPN group at the University of Aarhus, 2001) and JFern (Nowostawski, 2000), that enable users to construct and analyse systems by means of CPNs.

In CPNs, places may contain zero or more tokens. Arcs specify the schema that they carry. Specifically, arcs exiting and entering a place may have an associated expression that determines which tokens are to be removed or deposited (arc inscription). Simple Boolean expressions, called guards, are associated with transitions and enforce some constraints on tuple elements. In order for a transition to be fired, the transition guard must evaluate to true, taking into account the variable bindings. When a transition is fired, the tokens matching the criteria are removed from the input places, and expressions that specify the output arcs’ expressions are evaluated to construct the new tokens that are to be put in the output places. If the evaluation of an output arc returns null, then no token is placed for the given output place.

Figure 5.1 shows an example of a CPN model. In this example the transitions request resource A and request resource B may be fired concurrently. If both resources (A and B) are available, then the transition execute the task executes the task.
A CPN can be defined by a 9-tuple:

\[(\Sigma, P, T, A, N, C, G, E, I)\]

Where

- \(\Sigma\) is a finite set of non-empty type specifications, also called colour-sets. A colour-set is the same as a data type in computer languages, and it limits the values that can be assigned to an entity to some predefined values. These values are called colours. Colour-sets can be generally all the types that are available in a computer language. The colour-sets are defined with declaration statements similar to those used in high-level computer languages.

- \(P\) is a set of places. Places may contain zero or more number of marks called tokens. Each PN has input and output places. An input place is a place that is examined to see if a transition has been enabled, and an output place is a place to which tokens may be added when a transition fires. Transition firing may or may not add tokens to an output place.
- $T$ is a finite set of transitions. Transitions represent the actions of the system. A transition usually has a code segment (which is a block of code) associated with it. This code is executed when a transition fires.

- $A$ is a finite set of arcs. An arc connects a place to a transition, and has a direction which indicates whether the place is an input or output place. An input arc connects a place to a transition and an output arc connects a transition to a place.

- $G$ is a guard function. The guard function $G$ maps each transition $t$, to a boolean expression. In addition a guard can also be a list of boolean expressions. Intuitively, this means that the binding must satisfy all the boolean expressions.

- $E$ is an arc expression function. The arc expression maps each arc to an expression. Arcs exiting and entering a place may have an associated function that determines what tokens are to be removed or deposited.

- $I$ is the initialisation function.

### 5.2 Otago agent platform

Opal is a distributed software agent platform that complies with the standards provided by FIPA (FIPA, 2000c). Opal was described in Section 2.9.6. The modularity of Opal enhances its flexibility, and it has been extended in various works, some of which are $a$) the integration of CPN for specifying the interaction (or conversation) protocols (Purvis et al., 2003), $b$) an extension that specifies IPs by using Agent Unified Modelling Language (AUML) (Ehrler and Cranefield, 2004), and $c$) support for different high-level reasoning engines (Wang et al., 2005).

The Opal architecture also allows for agent abstractions at multiple levels. The lowest-level agents are known as micro agents, which represent “the lowest and most primitive level of agent instantiation”, and these are well suited for system-level programming tasks (Purvis et al., 2002). On the higher level, there are the FIPA-compliant agents, which have advanced capabilities such as using ontologies and performing reasoning on messages. The Opal agents are interpreted as “entities deployed in a multi-agent system” which are capable
of playing one or more roles in the agent society (Purvis et al., 2002). Micro-agents do not necessarily have their own thread of control and can even be reduced to pure reactive agents. A typical Opal agent could contain numerous micro-agents to perform tasks, such as dispatching messages, managing conversations, and executing plans. Importantly, each Opal agent has its own thread of execution, which allows for concurrent execution of multiple agents.

A critical feature of Opal is its support for the FIPA ACL specification for agent communication (in addition to the implementation of the necessary components of the FIPA abstract architecture). Opal additionally provides all basic Java Agent Services (JAS) \footnote{Java Agent Services. http://sourceforge.net/projects/jas/} and supports RMI, FIPA 2000, HTTP and IIOP as message transport protocols. In Opal all the inter-agent interactions are modelled as asynchronous messages, while all the intra-agent interactions are currently method calls on different modules within the agent. Our framework employs Opal agent platform with no major changes to facilitate asynchronous communication among agents.

5.2.1 Opal conversation manager

Our definition of conversation is similar to the one proposed by Nowostawski et al. (2001). They define a conversation as an ongoing sequence of messages that can span multiple agents and IPs. According to Nowostawski et al. (2001), a conversation can be decomposed into several layers. The basic layer is called the protocol layer. The protocol layer is the interaction protocol and provides a template of sequences of expected communicative acts that are based on the roles played by interacting agents. This definition is compatible with the FIPA definition of a protocol. According to FIPA, a protocol is “a common pattern of dialogues used to perform some generally useful task; the protocol is used to facilitate a simplification of the computational machinery needed to support a given dialogue task between agents; simply a dialogue pattern” (FIPA, 1997). Note that protocol layer is another level of abstraction in the system. We have not established any protocol because we used interaction protocol defined by FIPA. However, we have established our own conversation model described in this thesis.
Another layer is called the conversation layer which is a particular instance of a protocol or set of protocols. Therefore a protocol can be seen as a set of possible sequences of communicative acts, whereas a conversation in this context shows a particular sequence of communicative acts.

Opal offers a Conversation Manager (CM) module to manage the complexities of conversations in situations where each agent may concurrently interact with several agents and the decision of any agent in each conversation impacts the agent’s other conversations. Each Opal agent has its own CM to manage its conversations. The Conversation Manager uses CPNs in handling agent’s conversations. For any interaction with another agent, the CM creates a local instance of a conversation, which subsequently creates a CPN model and puts an initial token marking into its CPN model. The agent then executes this CPN, which leads to a message (or messages) being sent to the appropriate agents. The receiving agent receives the message through its CM and similarly executes its appropriate CPN model based on the new token that was received (the message). After the first interaction between two agents, if any of the agents receives a message from the other agent, its CM will find the corresponding “conversation” and resume it. This conversation process is the basis of agent communication in this distributed approach, where agents reside on different hosts. Thus use of CPNs enables agents to handle multiple concurrent conversations and facilitates the ability to resume a particular conversation when a relevant message arrives. Figure 5.2 shows a high-level view of the Opal system.

The original version of the CM allowed for the definition of two roles: initiator and participant. We have changed the original implementation of the CM so agents involved in each conversation can play several roles. Moreover, previously the CM was creating a thread for each CPN model of a conversation. However, JFern uses a loop that keeps executing the CPN until there is no transition to be fired (the enablement list is empty). Therefore there was no need for the thread, and it was removed.

5.2.2 JFern

JFern (Nowostawski, 2000) is a CPN development and simulation tool-kit developed at the Software Engineering and Collaborative Modelling Laboratory (SECML) of the University
of Otago. The JFern tool-kit provides a graphical user interface for constructing PN models. The CPN models are created using the JFern editor, which provides all the necessary CPN notations and editing functions. The resulting CPN contents (places, transitions, arcs declarations) are stored in an XML file for interoperability purposes and a layout file stores information about the coordinates of the PN graphical elements. Figure 5.3 shows a snapshot of the JFern editor.

The Opal CM module uses the JFern simulation engine to execute its conversations. In order for the CM to work with a CPN model, some special CPN places should be initially created in the model.

- **start** place. This place is only used for initialising the conversation (putting the token which includes the necessary information for the conversation, such as the conversation ID and the name of the IP). This place is only required for the initiator of a conversation.

- **in** place. All incoming messages to the agent are handled by the CM, and relevant information associated with a message will be encapsulated in a token and inserted directly into this place. This place is required for the initiator and participants in a conversation.

- **out** place. The **out** place contains all the outgoing messages. For sending a message,
Figure 5.3: JFern snapshot.

the CM constructs a message by using the relevant information of the token and sends it to the receiving agent over the network. The receiving agent will take the received message and insert the relevant information as a token into the in place of its appropriate CPN. This place is required for both the initiator and participant in a conversation.

Once a message is received into the in place of a CPN model, that model is executed. Depending on the information in the body of the message (and after the execution of the CPN), the agent may need to send some information to the requesting agent. In this case a message token that contains the essential information is produced and put into the out place of the model, which is then automatically sent to the requesting agent.

Throughout this thesis the in and out places of the CPN models are represented using double circles to differentiate them from other ordinary places of the CPN models.

Figure 5.4 shows how a message token may be sent from one agent to another. In this example agent a sends a request to agent b. The send request transition creates a message token and puts the token into out place. The CM of agent a sends the message to agent b. The CM of agent b receives the message and puts the token into the in place of agent b, and then the process request transition fires to process the message contents. For a more detailed
5.3 Modelling agents’ roles

Wooldridge et al. (1999) define roles from a software engineering perspective. They define a role as “an abstract description of an entity’s expected function”. Psychologists have further refined the definition of roles by differentiating between social roles, which “identify people by what they do and (or) their relationship to someone else”, and functional roles, which “describe what a person actually does with and for another person” (Darling et al., 2002). Gerkey and Mataric (2004) use the terms task and role interchangeably in a multi-robot context, since in their work “the underlying problem (of task and role allocation) remains the same”. However, they mention that a role usually has a time-extended implication, whereas a task is more transient in nature.

Here task refers to a piece of work that an agent or team of agents must perform. Our definition of role is similar to the one described by Wooldridge et al. (1999): an abstract description of agents’ responsibilities and expected functions.

5.3.1 Agents’ roles

In our disaster scenario example, agents with special equipment such as cameras and heat sensors are utilised to find victims. Moreover, different agents with different equipment are needed to rescue the victims. In that context, there is a need for at least two roles. The following example roles were defined for the agents in the framework:

- **Initiator**: The initiator role can be played by a task detector agent which is capable of identifying tasks by using some sensors (for instance, cameras or heat sensors). If an
agent detects a task, then it may start a conversation to find teammates for performing the task.

- **Helper**: The helper role can be played by an agent that has the necessary equipment (for instance, shovels and fire-extinguishing equipment). These agents can be recruited by agents playing the initiator role.

Roles are associated with agents that carry out certain tasks. The expected behaviour associated with a role is represented by a CPN model. The CPN for each role is designed based on the roles that agents could play in a conversation. In each conversation, each agent must first identify its role in that conversation. Once the role of the agent is determined, the agent can then create an instance of the CPN model associated with that role. Further information about the example roles used in this chapter is provided in Sections 5.3.3 and 5.3.4.

### 5.3.2 Timeout

All the CPN models designed for specifying roles use an environmental system time to accommodate the time delay and the lack of guaranteed message delivery associated with agents’ responses in a distributed multi-agent environment. The role specification must also consider a timeout mechanism in order to cope with situations where a response from an agent is not received or delayed. Consider a scenario where an agent exits a region and is therefore no longer in the communication range of the other agents in that region. An agent may also become unavailable to respond due to a fault in its communication module. In such situations, the agents that are expecting the response must have a timeout mechanism so they do not wait indefinitely for a response from an unavailable agent. All CPN models for different types of roles used in this work have a timeout mechanism to cope with such situations. An agent resumes executing its CPN if it receives all the responses or timeout occurs, whichever occurs faster. All messages received after the timeout are disposed.
5.3.3 Initiator role

Figure 5.5 shows the CPN model for the initiator role. The transitions are marked with alphabetical letters, which are referred to in this section (so the reader can easily follow the CPN model for this role).

(a) When an agent finds a task, it creates a token and puts it into the start place. The token has the task information (requirements, time constraint, etc.), the name of the IP, and the conversation ID for the conversation. The names of the helper agents in the neighbourhood are put into the neighbours place. The agent then sends “help” requests to its neighbours.

(b) After sending the requests, the sent time is put into the time place. The transition process timeout keeps checking the agent’s current time. If the difference between current time and the time that the message was sent is less than the agent’s waiting time for getting a response, then it puts the timed token back to the time place, otherwise it passes the token to the responses place.

(c) the transition receive responses receives all the responses and stores them in a CPN variable. After the waiting time elapses, the agent begins processing all the responses that it has received and selects its teammates.

Figure 5.5 shows a hierarchical view of the initiator CPN for the process received responses transition. The dashed lines in the figure represent the decomposition. The hierarchical representation is used to provide a clear description of the actions performed in the process received responses transition.

(d), (e) The transition collect acceptance responses collects all the positive responses, and the select teammates transition (d) processes the positive responses.

(g), (f) If the agent could find other helper agents with the required capabilities, then the team is formed. Once the team is formed, then a “move” message is sent to the selected teammates that directs the agents to move to the task location. This message also contains the details of the team. The agent also sends a reject message to all the agents who have responded positively but have not been selected as teammates, and informs them that they are not selected. If the agent cannot find agents with the required capabilities, then it sends a message to all the agents that have responded positively and informs them that the task is cancelled (this is done via the send reject message transition). In such a situation the initiator
drops its current task and starts searching for new tasks.

Figure 5.5: The CPN model for the initiator role.

5.3.4 Helper role

Figure 5.6 shows the CPN model of the helper role. The details of the CPN model of the helper role are described below (refer to the dashed lines surrounding the characters “a”, “b”, “c”, “d”, “e” in the diagram in connection with the following discussion).

(a) A helper agent receives requests from various initiators in its neighbourhood. If the agent is involved in performing another task, then it sends a reject message to the requesting agent (transition send reject). However, if the helper is available, then it may want to wait for a certain length of time to receive several requests and then select the best offer. The CPN model of the helper accommodates the waiting time by including the process timeout transition. After receiving a request, the helper agent puts the current time as a token into the time place. The process timeout transition compares the time that the first request was received with the current time. If the difference is more than the waiting time for helper agents, then the CM passes all the received requests to the process request transition, otherwise it
puts the time token back to the time place. When the waiting time has elapsed, the process request transition processes all the requests and selects the best offer. It also sends a positive response to the initiator and changes its status to unavailable.

(b) When a helper agent receives a move message from an initiator, it starts moving toward the task. The move message also contains the names of the teammates so the helper agents know their teammates. Moreover, the move transition estimates the agent’s arrival time (at the task location) based on its distance from the task location. The agents move by taking steps toward the task location, where each step refers to moving to the next cell of the grid environment. After each step the agent checks its current location to see if it has reached the destination. If the agent does not reach the task location before its estimated arrival travel time, then the agent sends a not reached task message to its teammates and informs them about its status (send not reached task transition). If the agent reaches the task location before the arrival time, then it sends a message to its teammates informing them of its current location (send reached task transition).

(c) The teammates process the location of the agents that have reached the task location. Once all the teammates become available at the task location they start performing the task. For performing a task, each agent has to spend some time at the task which in our work is set proportional to the fraction of the task that is to be completed by the agent.
If the agent does not reach the task location after the estimated arrival time, then it sends a message to its teammates informing them about the issue. In this situation, all the team members cancel their current contract (in the next chapter we address this issue in Section 6.3.3).

A helper agent with a positive response to a request may be rejected by the initiator of the conversation if better-suited helpers are available. In this case the rejected agent changes its status to available and can participate in other tasks.

5.4 Task

The task description in this chapter is slightly different to the task description provided in the previous chapter. This new task description includes a priority for tasks. A task is represented as a 4-tuple:

$$< R, t, p, w >$$

where

- $R$ is the set of requirements.
- $t$ is the time constraint of the task.
- $p$ is the task priority.
- $w$ is the basic task reward.

Tasks are produced by a task monitoring agent which produces tasks at certain time intervals, and removes old tasks from the environment. The tasks are distributed randomly in the environment, and their requirements are generated randomly at the run-time. The task time starts from the moment that a task is produced.

5.4.1 Task selection strategies

The following describes the strategies that agents could adopt for selecting their tasks. These strategies are applicable to emergency scenarios where new tasks with various urgency levels may appear at the different locations unpredictably. The aim is to study the effect of these
strategies on the performance of the agents with various teammate selection strategies and thereby determine which task selection strategies are most appropriate for different circumstances. Please note that these strategies are separate from teammate selection strategies.

**High priority task selection strategy**

Agents with this strategy select a task with the highest priority within their vicinity (task reading range). If there are several tasks with the highest priority value then the spatially closest one is selected.

**Nearest task selection strategy**

Agents with this strategy select the task that lies at the least distance away from them. If there are several tasks at the same distance, then the task with the highest priority is selected. In cases where there are several tasks with the same priority value, then the agent randomly selects a task.

### 5.5 Agent reward

In this chapter the reward equation considers the task priorities. Tasks with higher priority (urgency) generate a higher reward. The following equation calculates the reward in the current implementation of the framework.

$$\lambda = \sum_{i=1}^{n} \min\left(\sum_{j=1}^{m} \frac{(a_j)_{e_i}}{r_i}, 1\right) \times \frac{w \times p}{m}$$  \hspace{1cm} (5.1)

where

- $m$ is the number of participating agents.
- $n$ is the number of task requirements.
- $p$ is the task priority.
- $r_i$ is the $i^{th}$ requirement of the task.
\((a_j)_i\) is the capability of the agent \(a_j\) for \(i^{th}\) requirement.

\(w\) is the basic task reward.

5.6 Experiments

Various experiments were performed to examine and demonstrate the different aspects of the framework. The aim of these experiments was to ensure that the framework can be used for testing the cooperative behaviour of agents in realistic environments where multiple agents act concurrently. Particularly, we focused on the impacts of the spatial and temporal constraints on the agents’ performance. Agents can employ various strategies that consider the temporal and spatial constraints, thus enabling us to measure the impacts of these strategies on agents’ performance. We also show how agents can learn the best strategy. This is particularly useful when agents operate in dynamic situations and agents need to adapt to new circumstances.

The experimental agent framework was tested by employing multi-threaded Opal agents on a simulation grid-type environment where agents are capable of running multiple conversations over various tasks concurrently. All the messages among agents are asynchronous and therefore the messages are not received instantaneously. The simulation environment is a grid of 100 by 100 cells that contains 100 agents with a random initial distribution. 16 out of 100 agents were able to detect tasks (initiators), while the others were restricted to being helpers. The limited number of initiators reflects real situations in which there are a few agents with expensive task identifying sensors and equipment. The agents’ capability values and task requirements were generated randomly and normalised to range from 0 to 1.0.

Table 5.1 shows the experiment parameters. A single performance time unit refers to the time required to move from one cell of the grid environment to an adjacent cell. In addition, there is some time associated with an agent performing a simulated subtask. The subtask execution time is proportional to the fraction of the task that must be completed by the agent. The execution time for each agent depends on the size of the subtask that is assigned to the agent (it is the size of the subtask \(\times\) time unit). For instance, if there is a helper agent that has to clean up rubble that is as large as two square metres, then the agent must spend two time units to perform such task. Similarly, if a fire must be extinguished from an area as
large as two square metres, then the agent must spend 2 time units to extinguish the fire. Since carrying victims usually requires extra care, we assume that two time units is required for carrying each patient to a safe place. The experiments assume that a team can have an arbitrary number of agents that are selected by the initiators. It is also assumed that each agent has one capability that depends on the equipment that it carries.

Table 5.1: Simulation parameters. All time values are given in seconds.

<table>
<thead>
<tr>
<th>parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of agents</td>
<td>100</td>
</tr>
<tr>
<td>Task production interval</td>
<td>120</td>
</tr>
<tr>
<td>Waiting time for non-Impatient strategies</td>
<td>12 s</td>
</tr>
<tr>
<td>Waiting time for <em>Impatient</em> strategy</td>
<td>4 s</td>
</tr>
<tr>
<td>Waiting time for helpers</td>
<td>1 s</td>
</tr>
<tr>
<td>Time unit</td>
<td>3</td>
</tr>
<tr>
<td>Visibility range for agents</td>
<td>10 cells</td>
</tr>
<tr>
<td>Visibility range for agents with <em>Nearest available</em> strategy</td>
<td>6 cells</td>
</tr>
<tr>
<td>Visibility range for tasks</td>
<td>5 cells</td>
</tr>
<tr>
<td>Basic reward</td>
<td>5</td>
</tr>
<tr>
<td>Number of tasks</td>
<td>varies</td>
</tr>
<tr>
<td>Task priority</td>
<td>randomly generated integer values [0,5]</td>
</tr>
</tbody>
</table>

The visibility range of all the agents is fixed to 10 cells. This was selected based on the size of the grid and the number of agents. The visibility range for our tests must be relatively large so the agents can see a number of agents in their visibility field in order to have the option of selecting their teammates. Three visibility ranges were tested 4, 7 and 10 and the visibility range of 10 seemed to be reasonable (since it allowed agents to see a number of agents in their visibility range so they can choose among different agents). The *Nearest available* strategy was implemented by reducing agents visibility range (6 cells). This is due to the preference of these agents for selecting nearby agents. The agents visibility range for tasks is set to 5.

After the physical layout of the problem space and robotic agents was established, I
experimented with various time settings (associated with waiting times, timeouts, etc.) that would be appropriate for the application under consideration. That is, they were chosen so that they would enable the agents to complete their tasks in a reasonable amount of time. Slightly different parameters settings would have been plausible, and one could conceivably carry out an exhaustive study to determine the optimal parameter settings. However, the point of the experiments here is not to determine the optimal parameter settings but, instead, to demonstrate the practicality of the framework and how it can be used to comparatively evaluate different approaches for a given problem domain.

All the experiments are repeated ten times. In these experiments, teams of any size can be formed. Moreover, the experiments employ the teammate selection strategies introduced in Sections 4.3 and 4.6.3.

5.6.1 The effect of teammate selection strategies on agents reward when task density is low

The aim of this experiment is to show the effect of various teammate selection strategies on agents rewards in situations where task density is not very high and under various task time constraints. In this experiment the task manager produces 20 new tasks at 120 second intervals and removes the old tasks that have not been performed. Since the simulation grid size is relatively large and the number of tasks is very low, the agents occasionally find no tasks to perform. All the tasks in this experiment have the same priority value (1).

Four groups of initiator agents were employed in each simulation run. The initiators employ the model described in Section 5.3.3. Each group employs a different teammate selection strategy, so various strategies compete against each other. All initiator agents wait for 12 seconds to receive their responses, except impatient agents which wait 4 seconds. The helper agents employ the model described in Section 5.3.4. The experiments were run for 1200 seconds (20 minutes).

Figure 5.7 shows that when the task time is very constrained (30 seconds) the Impatient strategy outperforms other strategies, because agents with this strategy wait for a short time (and do not wait to receive responses back from most of the requested agents), thus having a higher chance of performing their tasks. However, agents with other strategies may not
Figure 5.7: The effect of strategy on reward when task density is low. The average reward is the reward of all the agents that follow the same strategy averaged over 10 runs.

be able to reach the task positions and perform the tasks before the time expires (since they wait for a longer period). When the task time is less constrained (50 seconds), the *Nearest available* strategy outperforms the other strategies. These agents wait for a given length of time based on waiting times in Table 5.1 and select the closest agents as their teammates to work on tasks. Since agents with this strategy select the closest agents, they have a higher chance of performing their tasks within the given time. The low overall reward of agents with the *Impatient* strategy is due to teams performing only small parts of those tasks (since they do not wait long enough to find agents with the required capabilities). The reward of the *Best available* strategy is less than the reward of agents with the *Nearest available* strategy under somewhat more relaxed time constraints (50 seconds). Agents with *Best available* strategy selected teammates with high quality capabilities which may be farther away from the task position and therefore they may not have enough time to reach the task position within the specified time. When the task time is further relaxed (70 and 90 seconds), the *Best available* strategy starts to outperform other strategies. This was due to their performing larger portions of the tasks by having high quality teammates. The agents with the *Best possible* strategy performed worst under all tested time constraints. This is the effect of the perfectionist attitudes of these agents. This approach would be valuable, though, when there
is more incentive in fully completing a job.

5.6.2 The effect of teammate selection strategies on agents reward when task density is high

The previous experiment was repeated in this test to study the effect of agents’ strategies on their rewards when the task density is high. Here the agent manager produces a high number (200 tasks) of new tasks at 120 second intervals.

Figure 5.8 shows that when the task time constraint is very tight (30 seconds), the result is similar to the previous experiment: the agents with the Impatient strategy generally outperformed the other strategies. However, as the task time starts to relax (50 seconds), the Nearest available strategy outperformed the other strategies. This trend continues when task time was even more relaxed (70 and 90 seconds). This is due to the fact that agents that select their nearest neighbours reach the task location more quickly. Since the task density is high, these agents will perform more tasks and therefore increase their rewards. In addition, the reward of agents with the Best available strategy is less than the reward of agents with the Nearest available strategy across all tested task time. This is due to the fact that agents with the Best available strategy select their teammates based on the quality of their capabilities. Such agents may be farther away from tasks. Thus it takes longer for the teammates to move to the task location.
The average reward is the reward of all the agents that follow the same strategy averaged over 10 runs.

The reward of the agents with the *Impatient* strategy was lower than the reward of the agents with the *Nearest available* and *Best available* strategy under less restrictive task time constraints (50, 70, 90 seconds). This occurred since the Impatient agents did not wait long enough to receive their response back from the requested agents. Therefore they usually formed a team with a limited number of agents which in most cases did not have the required capabilities. Such agents are only capable of performing a small part of each task.

5.6.3 The effect of agent learning on agents reward

In our framework, agents are required to change their strategies under various circumstances in order to adapt to dynamic conditions in the environment. Therefore there is a need for mechanisms that allow agents to find a strategy that improves their reward for the agents under varying circumstances. This experiment demonstrates how the agents individually can identify the best strategy for the agents’ current circumstances.

In our experiment a simple learning mechanism was adopted. Agents (both initiators and helpers) were assigned an initial strategy. In this experiment the agents’ roles were fixed although the framework allows agents to dynamically change their roles (a scenario requiring agents to alter their roles is studied in the next chapter). Agents periodically assess
and record their performance. The agents start operating with that strategy for a certain period. This period must be long enough to give the agents a chance to perform a few tasks. After this period the agents select another strategy and repeat the aforementioned process.

Each agent builds a list of different strategies and updates the average reward received for each strategy after a task is performed. After exploring different strategies, each agent makes a decision based on the gained reward for each strategy and selects the strategy that has given the agent the highest reward. After selecting the strategy, the agent operates according to its selected strategy for another round. If the current reward (gained during the current period) is less than the previous reward, then the agent starts exploring its strategies again, otherwise it keeps performing with its current strategy for another period. This simple learning mechanism allows agents to adapt to dynamic conditions in the environment.

For this experiment two simulation runs were executed. In one run four groups of agents were formed, where each group employed a single teammate selection strategy. The strategy of the agents was fixed for this experiment. In the second run all the agents were set to be learning agents and were able to change their strategies if the following condition held:

\[ R_c - R_p < \alpha \]

Where

- \( R_c \) is the reward that has been achieved during the current performance period.
- \( R_p \) is the reward that has been achieved during the previous performance period.
- \( \alpha \) is the learning threshold.

The agents (both initiators and helpers) selected a strategy and operated according to that strategy for a certain period of time (240 seconds, we refer to this as exploration period). After this period, if the agent had not had a chance to perform a task with that strategy then it continued with its current strategy. However, if the agent had not been able to perform a task after 3 exploration periods, then the reward for that strategy was considered to be 0, and the agent changed its strategy. Since experimental conditions are critical in choosing time related parameters, the exploration period and (helper agents) waiting time were selected long enough to allow experiments proceed realistically. By the end of the exploration phase (the first 960 seconds), most of the learning agents had selected their strategies. Since the
task density was high, and task time constraint was relaxed, the learning agents changed their strategies to *Nearest available*, which allowed agents to perform a high number of tasks and therefore improved their reward. Figure 5.9 compares the total reward of agents in the system when agents learn and do not learn (in the next chapter, the effect of agents adaptation in a more complex dynamic situation is studied).

![Figure 5.9: The effect of agents learning. The average reward is the reward of all the agents in the system in learning and non-learning scenarios averaged over 10 runs.](image)

In real life scenarios, task arrival rate and urgency of task completion change dynamically. This experiment shows that agents in our framework can dynamically modify their strategies based on environmental changes.

### 5.6.4 The effect of agent task selection strategy on agents reward

This experiment investigates the effect of the task selection strategy on agents reward. The number of tasks in this experiment must be high in order to provide some task options for the agents. In addition, the task time must be less constrained in order to observe the impact of the agents’ task selection strategies. The task manager produces 200 new tasks at every 120 second intervals, so task density is high. Moreover, the task time constraint is very relaxed (90 seconds).
Figure 5.10 shows that *Best available* strategy performed better when agents selected high priority tasks. These agents performed large fractions of the tasks and since the tasks are of high priority, their rewards improved. The agents with the *Best possible* strategy performed a small number of tasks due to their perfectionist attitude. Therefore the impact of selecting the high priority tasks was not great. The agents with the *Nearest available* strategy were able to perform more tasks by selecting the nearest tasks. Therefore the reward of these agents was not greatly improved by selecting the high priority tasks. High priority tasks may have higher rewards but it took longer in general for the agents to reach the task locations. The *Impatient* strategy performed well by selecting high priority tasks which gave the agents a chance to gain a high reward.

![Figure 5.10: The effect of agents task selection strategies on agents reward.](image)

5.7 Summary

This chapter introduced some extra features to the cooperation framework. The framework now employs the Opal agent platform and its Conversation Manager module. The use of multi-threaded Opal agents allows the agents to communicate using asynchronous messages. Asynchronous messages further enhance agents autonomy, since they allow agents to continue operating instead of waiting to receive their responses. In addition, Opal agents communicate with each other using the standard FIPA protocols. This feature enables our agents
to communicate with FIPA-compliant agents.

The framework employs Coloured Petri Nets to model concurrent interactions of the agents. Coloured Petri Nets were useful for modelling and managing multiple conversations for individual agents. This facilitates the development and use of agent-based systems in real time situations in which the agents must coordinate with other agents in a concurrent fashion. The framework enables the agents to execute their CPN models by using the JFern engine. Although we only presented two roles for the agents in this chapter, the framework allows the use of multiple roles for each agent. In the next chapter, we demonstrate how agents can change their roles if required. It also accommodates dynamic situations in which the participating agents may change their strategies in response to past experience if they can learn from their past experiences.

While the developed framework was empirically examined here by performing computer simulations of agent activities, the framework is ultimately intended for fast design and test and then deployment on real, physical robots. It is with this ultimate goal in mind that our FIPA-compliant multi-agent simulation framework has been developed to come close to reproducing realistic situations by including the concurrency conditions of real distributed multi-agent robotic systems.
Chapter 6

Modelling Tasks

In the previous chapter we explained how the CPN modelling technique was employed in our framework to facilitate coordination of cooperative activities among agents. Agents may have different CPN models that suit different activities, and they coordinate their activities according to their models. An agent determines which model to use depending on the IP (that specifies the type of the activity that the agent engages) and the role that it plays in each activity.

This chapter introduces a model for defining general dependencies between subtasks (or requirements) in our own framework. It introduces a CPN model for tasks and the coordination mechanism required to handle the dependencies of complex tasks in our tightly integrated collaborative framework. We refer to this version as the ‘comprehensive’ framework. The use of CPNs is advantageous since they can be directly mapped to execution via the JFern execution engine (Nowostawski, 2000). Our approach is similar to Raposo and Fuks (2002), since it considers temporal constraints among the task requirements. However, we consider a distributed environment where various agents interact over a network. Thus task coordination is achieved by analysing received messages and updating the task CPN dynamically. This chapter provides the details of our modelling technique for tasks and its implementation. In addition, it provides results of more experiments that show how agents can adapt under changing circumstances.

Our example scenario includes complex tasks in which the requirements must be satisfied with different capabilities of agents. So far we have assumed that when a team is formed
all team members go to the task location, and only when they all arrive can the task be performed. However, this is not in alignment with some realistic situations where temporal dependencies between subtasks establish the order of execution for subtasks. If there are no dependencies among subtasks, then there is nothing to coordinate in a collaborative activity (Malone and Crowston, 1994). Therefore there is a need for mechanisms in our framework that allow managing and coordinating dependent subtasks, and ensuring that dependencies are not violated (Raposo, Magalhães, Ricarte, and Fuks, 2001). Coordination is important especially where the resources for performing tasks are distributed.

A large number of mechanisms have been proposed to facilitate the coordination of complex and collaborative tasks. Previous work in this area was restricted to specific scenarios that employ strictly defined protocols. Such systems are not generic and adaptable, as the protocols are hard coded, thus limiting them to predefined scenarios (Flores et al., 1988; Fuks et al., 1995). Another group of coordination mechanisms allows the agents to select their coordination mechanism at run-time instead of embedding them into the system (Malone et al., 1995; Schmidt and Simonee, 1996). For such mechanisms to be useful in dynamic situations, they must flexibly allow dynamic and temporary changes to the coordination schemes.

Coordination languages exemplify the second category (Cortes, 1999; Ciobanu and Lucanu, 2005), and they have been used to facilitate coordination of concurrent processes. They are also useful in the construction of collaborative applications. These coordination languages have similar syntax to programming languages. However, coordination languages operate at a low level of abstraction and do not provide or support verification mechanisms. Petri Net-based techniques represent another examples of the second category that have been used for modelling tasks. PN allows for specifying the temporal and resource-related dependencies among subtasks. In addition, it allows for modelling concurrent actions performed on tasks. Moreover, PN based models can be directly mapped to executable code. These features make PN a desirable technique for modelling complex tasks. The work of Huang and Zhou (2005) and Raposo and Fuks (2002) are examples of such systems. Huang and Zhou (2005) used PN for coordination of interdependent tasks in the context of Computer Supported Cooperative work, but their work does not specify the actual implementation and execution details of their modelling method. Raposo and Fuks (2002), and Raposo, da Cruza, Adrianoa, and Magalhães (2001) used PNs for specifying generic interdependencies among
tasks in tightly collaborative activities. Their coordination model considers temporal and resource-related interdependencies. Their system uses a central coordinator component that is in charge of coordinating the execution of tasks. Although they consider a distributed system, it is not clear how the distributed entities (agents) communicate with each other.

6.1 Task specification

Here a task is represented by the following 3-tuple:

\[< R, t, w >\]

where

- \( R \) is the task requirements list. In our disaster scenario, the agents may use heat or motion sensors in addition to cameras in order to determine the task requirements (i.e., the number of victims, presence of fire, and so on). This is done by the initiator agent, which has the equipment (i.e., sensors) for estimating the task requirements. The task requirements specify the task type. Each task type is associated with a particular task CPN model. The details of the task CPN model are explained in the next section. The task CPN model represents the dependencies among subtasks, in addition to the required capabilities for each task.

- \( t \) is the time constraint of the task. The time constraint indicates the time given to the agents to perform the task, and it starts from the moment at which a task is produced.

- \( w \) is the basic reward for the task.

6.2 Task CPN

The framework employs CPNs to model tasks. A task manager role is defined to coordinate the task execution. An agent playing the task manager role updates and maintains the task
CPN in consideration of the completion of subtasks performed by the participating agents (the task manager role is defined in Section 6.3.2).

Figure 6.1 is an example of a task CPN for the rescue scenario. As shown in the figure, the task has two phases. In the first phase an area of 8 square metres must be cleaned of rubble (clean=8). Moreover, fire fighter agents must extinguish a fire that has spread to an area of 6 square metres (extinguish fire=6). Once the rubble is cleaned up of rubble and the fire is extinguished, then the second phase can be started. In the second phase the agent(s) with carrying equipment must carry two victims to the nearest safe place (carry=2).

The CPN in Figure 6.1 indicates that the cleaning and fire extinguishing can be performed concurrently. Therefore once all the teammates for performing these subtasks (cleaning rubble and extinguishing fire) become available at the task location (available teammates place), then the assign requirements transition is fired. This leads to a message being sent to all the helper agents assigned to the first set of subtasks, asking them to start performing their assigned operations. Once the helper agents perform their subtasks, they inform the task manager. The task manager updates the performed places (for instance, the clean performed place). The task manager checks the result to evaluate the performance of the helper agents (collect results transition). This process is repeated for different phases of the tasks until all subtasks are performed. More details about the task manager roles are provided in Section 6.3.2.

The task CPN has several features in addition to the sequence execution for subtasks and enabling concurrency. The task CPN can reduce redundancies by taking into consideration which agent(s) is required at what point during the task execution. For instance, consider a task that has three phases. In the first phase, the cleaning agents may clean the area. In the second phase, the fire must be extinguished. In the third phase, more cleaning must be done and victims are rescued concurrently. In this example, the agent(s) that clean during the first phase of the task can perform the cleaning that is required at the third phase of the task as well. Therefore the task CPN reduces redundancies. Another advantage is that the task CPN enables the system to release the agents quickly after they perform their subtasks. The task CPN enables the task manager agents to identify whether or not a particular helper agent is required after it has performed a subtask. If the agent’s capability is no longer required, then it is informed so that it can participate in other tasks.
Figure 6.1: An example of a task CPN. The grey rectangles show the actions that are executed by the participating helper agents.
6.3 Agent models

The ‘comprehensive’ framework requires an additional role for managing tasks. This section introduces the various roles required for managing and performing the tasks. The initiator and helper roles were defined in the previous chapter. However, some modifications have been made to accommodate task management.

6.3.1 Initiator role

This new initiator role is similar to the earlier initiator role described in Section 5.3.3, but it has been adjusted to accommodate task management. As mentioned before, the initiator role is played by an agent that has sensors to identify the various requirements of a task. The initiator categorises the tasks into various types based on their requirements. For instance, a task that has victims under the rubble in an area that is on fire is of a different type to similar tasks that do not involve fire. The task type information enables the initiators to select the appropriate CPN model associated with each task. The task CPN model is then initialised with the task requirements. This CPN may be sent to the agent that manages the task in situations where this agent does not have the appropriate CPN model.

Once an initiator detects a task, it initialises its CPN model with the task information and the name of the IP. The name of the IP specifies the problem domain, and refers to the type of activity that the agent is undertaking (i.e., “rescue” in the case of our example scenario). The IP enables the agents to choose the appropriate role model from their role repository (agents can play different roles in different domains). It also initialises the model with the names of the neighbouring agents. It can be observed from Figure 6.2 that the initiator may send a request to one of the helper agents in its region and ask whether the agent is available to manage the task (see transition marked ‘a’ in the upper left of the Figure 6.2). If the helper agent rejects the request, then the initiator sends a request to another helper agent (b). This process is repeated until a task manager is found or there is no agent in the neighbourhood who can manage the task. In the latter situation, the initiator drops the task and moves to another area to find tasks. There may be some situations in which an initiator decides to manage a task, for instance where there are not many tasks in an area (initiator’s visibility field). In such circumstances, the initiator agent changes its role to task manager and starts
managing the task.

Figure 6.2: The CPN model for the initiator role.

6.3.2 Task manager role

The task manager agent recruits a set of helper agents to work on a task and monitors the task execution. Moreover, the task manager decides how the task execution should proceed. For instance, it decides whether a particular subtask can be performed by several agents as opposed to a single agent. Figure 6.3 shows the CPN model of the task manager role. The task manager is in charge of forming a team and updating the task CPN based on the messages received from the helper agents in the distributed framework. The details of the task manager role are described below (refer to the respective portions of Figure 6.3 outlined by dashed lines for the following discussion).

(a) The task manager agent initially reads the task requirements by traversing the task CPN. It then sends requests to all available helper agents in the its region (agents in its visibility range) and asks whether they are interested in performing the task. The helper agents decide whether to perform the task based on their strategies and availability.

(b) The task manager waits for some time for responses or until timeout occurs, and then it starts selecting its teammates. After selecting teammates, the task manager sends confirmations to the teammates and instructs them to move to the task location. The task
Figure 6.3: The CPN model for the task manager role.
manager will also send a reject message to agents that have responded positively but were not selected.

(c) The helper agents are given a time limit for moving to the task location (this time is the estimated time required by the farthest agent to reach to the task location and starts from the moment that the move message is sent to the helper agents). Each helper agent that arrives at the task location informs the task manager of its arrival. If the task manager receives a response from all the requested agents or the time allowed for agents to move elapses, then it starts updating the task CPN by putting the names of the available agents in the available teammates place in its task CPN. This leads to the execution of the task CPN. Once the first set of subtasks in the task CPN are assigned to helper agents, then the assign requirements transition of the task CPN (refer to Figure 6.1) updates the subtasks assigned place of the task manager model. This then sends a message to the assigned agents asking them to perform their subtasks.

(d) A timeout mechanism is used to handle situations where helper agents encounter problems and cannot perform their subtasks. Once the task manager receives the result of the task performance from the helper agents or timeout occurs, it updates the corresponding performed place in the task CPN. After the completion of each part of the task, a message is sent to those agents that are not required for the rest of the task, which frees them from their current commitment. This process is repeated until all subtasks have been performed.

6.3.3 Adaptive task manager role

The task manager role defined in the previous section does not allow the agents to adapt to unforeseen situations. For instance, there may be situations where a helper agent is not able to move to the task location due to some sort of fault, or it may be able to move to the task location but is not able to perform the task due to a mechanical fault in its equipment (for instance, a broken excavator bucket). If such a situation occurs when agents employ the task manager role defined in the previous section, the agents drop their current task and cancel their contract. This can be avoided if agents have a mechanism to deal with such circumstances.

Figure 6.4 represents the CPN role for a task manager that enables agents to replace
missing resources (i.e., agents with different capabilities). This CPN includes the recruit for missing teammates transition \((e)\). If, for some reason, recruited agents do not show up after a certain time (i.e., timeout occurs) then this transition enables the task manager to find other helper agent(s) that could perform the task. Moreover, if the helper agents show up but do not perform the task, this transition enables the task manager to replace the faulty agent. The task manager only cancels the task if it cannot replace the missing agents with agents that have similar capabilities.

### 6.3.4 Helper role

The helper role discussed in this chapter is slightly different from the helper role defined in Section 5.3.4, since it considers the requirements for task management. The helper agents are recruited by agents playing the task manager role. An initiator agent may decide to delegate the task management job to a helper agent in order to focus on finding other tasks in the environment. Figure 6.5 shows the CPN model for the agents playing the helper role. Refer to the dashed-line sections in Figure 6.5 for the following discussion.

\(a\) If a helper agent receives a request from an initiator agent to manage a task and it is not involved with another task, it accepts the request and changes its role to task manager. Otherwise it rejects the request.

\(b\) If a helper agent receives a request to help with a task and it has already committed to another task, then it rejects the request. This action supports the spatial consideration inherent in mobile robot task activities and takes into account the fact that each agent can only participate in one task which is located at a particular location and would be unable to perform two tasks concurrently. However, if the helper agent is not committed to another task then it accepts the request.

\(c\) A helper agent that is committed to a task may receive a move message. This message informs the agent to move to the task location. An agent that receives this message starts moving towards the task location. In this framework, there is a time limit for agent to travel to the task location. This time limit is the agent’s estimated time to travel to the task location and is calculated based on the agent’s distance from the task. This mechanism also allows the agent to inform its task manager in case the agent gets stuck in a location due to an obstacle.
Figure 6.4: The CPN model for the adaptive task manager role.
Figure 6.5: The CPN model for the helper role.
or some form of fault in their actuators. If the agent manages to reach to the task location within the expected time, then it informs its task manager that it has arrived. It also sends a message to inform the task manager if it fails to reach the task location in time.

\((d)\) Agents that have volunteered (responded positively) for performing a task may still be rejected by the requesting agent. In such situations, the helper agent changes its status to ‘available’.

\((e)\) An agent that is committed to a task may receive a message from the task manager to perform the task. The helper agents spend some time at the task location to perform a task. This time is proportional to the size of the subtask assigned to each agent. Once the task is performed, the agent sends the a “status report” back to the task manager.

6.4 Adaptation

In dynamic situations, agents are required to adapt to new conditions. We tested the adaptability of our proposed system in two types of situations. In one experiment, the number of tasks varied during the experiment and the task manager and helper agents were required to identify the best strategy under low and high task density situations. In another experiment, a portion of the helper agent population was unreliable, and the task manager agents were required to identify the unreliable population of helper agents and avoid interacting (forming teams) with them.

6.4.1 Adopting the best strategy

The adaptation mechanism must enable the agents to employ a strategy that maximizes the rewards under changing circumstances. Our adapting mechanism allow the agents to choose among various agents (both helpers and task managers) instead of simply accepting every request.

For selecting the best strategy, each agent creates a table that contains the strategies and their associated reward (which are set to 0 at the beginning). Agents update their tables by using a reinforcement learning mechanism (Sutton and Barto, 1998) after each task is performed. In the beginning, each helper agent randomly selects a strategy and only accepts
requests from agents with that particular strategy. The agent continues performing with its selected strategy for a certain period of time (exploration time). Then the agent selects another strategy and performs tasks for the same amount of time. This process continues until the agent has explored all the different strategies. If a helper agent becomes a task manager, then it retains its current strategy. Once an agent has explored all the strategies, it ranks them based on their associated rewards. From this point onward the agent ranks all requests based on the requesting agents’ strategies and selects the best strategy based on its own performance history. If the agent notices any changes to the environmental conditions, such as a change in task density, then it starts another exploration period to identify the strategy that suits the new situation.

### 6.4.2 Adapting to unreliable situations

Agents in open and dynamic environments may sometimes not honour their agreed upon commitments when performing tasks. In particular, this issue may arise in robotic scenarios due to mechanical faults. For instance, an agent with a damaged excavator bucket may be unable to clean rubble. Therefore agents require a mechanism for identifying the reliability of the other agents in the system. Here the reliability value refers to the ratio of the quality of the delivered capability to the committed quality. For instance, if an agent commits to clear 2 square metres of rubble from an area but only clears 1 square meter, its reliability value for its current performance is set to be 0.5. If on the other hand, an agent delivers its committed capability then that agent is regarded as completely reliable.

Each agent has a table for the reliability values of the other agents with which it interacts. Every time that an agent interacts with other agents for performing a task, it updates its reliability table using a reinforcement learning technique (Sutton and Barto, 1998). In addition, it stores the time that the update occurred. The update time is useful for task managers so they consider the most recent reliability values. After performing each task, all the teammates send their reliability tables to the task manager, and the task manager updates its table based on the teammate experiences and its own current experience. The task manager only considers the most recent performances of the helper agents when updating its reliability table (old records are removed). This updated model is then sent to all teammates so that they
also have an updated reliability table. Since we are dealing with a team scenario, we assume that all the agents are truthful in sharing their reliability tables with other agents. When a task manager decides to select teammates for a task, it ranks all the teammates based on their reliability values and then selects its teammates.

Figure 6.6 demonstrates an example of how agents learn about the reliability of other agents. In this example, agent $a$ is the task manager, and manages a task that involves agent $b$ and agent $c$. Figure 6.6(a) shows the reliability tables of agents before the task is performed. For this task, agent $c$ has committed to clear 4 square meters of rubble but only has performed 1 square meter, and agent $b$ has committed to clear 2 square meters and has performed according to its commitment. After the agents perform their tasks, they send their reliability tables to the task manager (agent $a$). The task manager considers the most recent reliability values for updating its reliability table. In this example we assume that the learning rate for the reinforcement learning is 1.0, and the time of performing the current task is 21. Figure 6.6(b) shows the reliability tables of agents after the task manager updates the values.

### 6.5 Experiments

This section provides the results of some experiments that were carried out using the framework. The prototype implementation was tested by employing Opal agents, with each agent running in its own thread and therefore all agents running concurrently. In addition, each agent is capable of running multiple conversations for various tasks. Each conversation is represented by a CPN and therefore multiple conversations were run concurrently. The framework also uses the CPN to present the tasks that need to be performed. The task CPN facilitates the coordination and management of the temporal dependencies between subtasks.

The experiments in this section were designed to show how the agents in the framework can adapt according to changing circumstances. In these experiments the agents can change their roles dynamically. In particular, helper agents can change their role to task manager during run-time if required. We also demonstrate how the CPN technique can facilitate agent management in unforeseen situations (i.e., unreliable situations).
Figure 6.6: An example of how agents’ reliability tables are updated.
6.5.1 Simulation parameters

We assume that agents have different capabilities that are useful in satisfying different task requirements. The agents’ capability values and task requirements were selected randomly and normalised to range from 0 to 1. The simulation environment uses a $50 \times 50$ rectangular grid of cells. We decreased the size of the simulation grid in order to increase the frequency of interactions among agents, thus the result of the adaptation mechanism can be seen in a shorter period of time.

In these experiments there were 60 agents with different capabilities, with 16 of them able to detect tasks and the remaining ones being helpers. Table 6.1 shows the simulation parameters for this experiment.

Table 6.1: Simulation parameters. All time values are given in seconds.

<table>
<thead>
<tr>
<th>parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Initiator agents</td>
<td>16</td>
</tr>
<tr>
<td>Number of helper agents</td>
<td>44</td>
</tr>
<tr>
<td>Task production interval</td>
<td>120</td>
</tr>
<tr>
<td>Waiting time for Impatient strategies</td>
<td>4 s</td>
</tr>
<tr>
<td>Waiting time for non-Impatient strategies</td>
<td>12 s</td>
</tr>
<tr>
<td>Waiting time for helpers</td>
<td>1 s</td>
</tr>
<tr>
<td>Visibility range of agents</td>
<td>8 cells</td>
</tr>
<tr>
<td>Time unit</td>
<td>3</td>
</tr>
<tr>
<td>Basic reward</td>
<td>5</td>
</tr>
</tbody>
</table>

All times in the table are listed in seconds, and this was measured by using the system time. A single performance time unit refers to the time required to move from one cell of the grid environment to an adjacent cell. In addition, there was some time associated with performing a task. New tasks were produced and old tasks were removed in every task production interval. The task time constraint was set so that the agents generally had enough time to perform the assigned tasks. When a task was completed, the task manager agent rewarded its agent team based on the equation 5.1. The task model used for these experiments was as described in Section 6.1, but with randomly generated values for its
requirements. Since the reward is proportional to the performed part of the tasks, it is a measure of the amount of tasks that have been completed. All the results presented here are averaged over ten runs.

6.5.2 Adopting the best strategy

This experiment was designed to show the adaptability of the agent system in dynamic environments where the number of tasks changes over time. It compares (a) a situation in which agents have fixed strategies (non-learning), with (b) a situation in which all the agents in the system are adaptive agents and can change their strategies according to the circumstances (learning). It also shows how the learning agents adapt their strategy under high and low task density circumstances. In the non-learning experiment, we had equal numbers of agents for each strategy.

In the learning experiment the agents started exploring with various strategies, using the adaptation mechanism described in Section 6.4.1. The task managers employed the model defined in 6.3.2. Agents observed their surroundings after taking a step (moving to the next cell). If the number of tasks in the region was more than 4, then they assumed that the task density was high. Otherwise the task density was assumed to be low in that region. The task time constraint was set to be very relaxed (100 seconds).

In this experiment the number of tasks changes every 32 minutes. At the beginning the number of tasks is high (100 tasks), and after 32 minutes it drops to smaller number of tasks (25 tasks). According to Figure 6.7, it took almost 24 minutes for the agents to explore with their different strategies under the high and low task density scenarios. After the exploration period, the learning agents were capable of identifying the best strategy by ranking their strategies based on the task density and the associated reward. From this point onward the agents selected their teammates, based on their experience.
Figure 6.7: The effect of learning the best strategy. The average reward is the reward of all the agents in the system in learning and non-learning scenarios averaged over 10 runs.

Table 6.2 shows the percentage of agents with different strategies in the learning experiment at point P1 and P2, which represent high and low task density situations, respectively. It shows that when the task density was high then the preferred strategy was *Nearest available*, which promotes the performance of more tasks by selecting closer teammates. The *Best available* strategy was less successful in environments with a high task density, because agents with this strategy select agents with a high quality of service, but they may be further away from the task. Therefore it takes longer for the high-quality teammates to move to the task locations, and therefore fewer tasks are performed. When the task density is low, then the preferred strategy was *Best available*, which performed a larger portions of tasks. The least preferred strategy was *Best possible*, whereby agents do not even recruit a team at all, unless they find helpers with capabilities for full task completion.

<table>
<thead>
<tr>
<th>strategy</th>
<th>Nearest available</th>
<th>Impatient</th>
<th>Best available</th>
<th>Best possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reward P1</td>
<td>76%</td>
<td>9%</td>
<td>13%</td>
<td>2%</td>
</tr>
<tr>
<td>Reward P2</td>
<td>20%</td>
<td>14%</td>
<td>63%</td>
<td>3%</td>
</tr>
</tbody>
</table>
6.5.3 The effect of agents learning about other agents’ reliability

The reliability-learning experiment was designed to examine how the system identifies reliable agents by employing the learning mechanism described in Section 6.4.2. All the agents were initialised with the Best available strategy, and 25 tasks were produced in each task production interval. In the first phase (first 24 minutes) of the experiment, 20%, of the agents were set to be unreliable and in the second phase, 40%. An unreliable agent is assigned a random failure probability generated at the time of its initialisation. Here the task managers rank the helper agents based on their reliability values. Agents for whom the task manager does not yet know their reliability values may be selected and given a trial for future reference.

We executed three different simulation runs for comparing three different scenarios. These scenarios include learning about the reliability of other agents (“learning”), learning about the reliability of other agents as well as adapting to unforeseen situations (“learning-adaptive”) by employing the CPN model explained in Section 6.3.3, and scenarios where agents do not learn (“non-learning”). The learning agents in this experiment employ the CPN presented in Figure 6.3 and cannot deal with unexpected situations (i.e., when agents do not commit to their contracts). In the learning-adaptive experiment, task manager agents employ the CPN model introduced in Figure 6.4, which enables the task managers to deal with situations when teammates fail to deliver their committed capability. These agents were capable of replacing the missing skills by recruiting new agents.
Figure 6.8: The effect of learning the reliability value of agents. The average reward is the reward of all the agents in the system (across different strategies) following non-learning, learning-adaptive and learning scenarios, averaged over 10 runs.

Figure 6.8 presents the results. In the learning scenarios agents initially start trialing other agents to learn their reliability values, and after about 12 minutes, the learning mechanism starts benefiting the agents. After 24 minutes from the start of the simulations, the population of the unreliable agents was increased by another 20% percent. This caused the reward of the agents to drop because it took some time for the agents to learn about the new, unreliable population. However, once the agents learnt the new reliability values their rewards again started to increase.

In this experiment the difference between the reward of the the learning-adaptive and learning experiment is greater when agents have not learnt the reliability values. This is due to the fact that in such situations agents may team up with unreliable agents which will not provide their committed capabilities. In those circumstances, the learning-adaptive scenario allows for replacing such agents and therefore improves the reward.

In order to see if the difference between learning and learning-adaptive is significant, we ran a $t$-test analysis (paired two samples for means). The $t$-test was run after the point where the agents learnt about the reliability values of other agents. This refers to the period between 12 and 24 minutes in the first phase (20% of the population is dishonest), and the period between 36 to 48 minutes in the second phase of the experiment (when 40% of the population is dishonest). There is a significant difference between means in the learning and
learning-adaptive (mean difference =7, \( p \) value=0.001, 4.7 < confidence interval 95% <9.3) when only 20% of the population is dishonest. When 40% of the population is dishonest the difference between means of learning and learning-adaptive is even more significant (mean difference =6.5, \( p \) value=0.000, 4.9 < confidence interval 95% of <8.1). The \( p \) values for both situations suggest that the difference between the performance of learning and learning-adaptive groups are highly significant (the range of \( p \) values varies between 0.00 to 0.05 where 0.00 shows high-significance and 0.05 weak significance).

6.6 Summary

This chapter has introduced additional new features to our framework. The task CPN formalism supports mechanisms for agents to identify task requirements. It also enables the task managers to manage the temporal dependencies among various subtasks. Moreover, it enables the agents to represent concurrent subtasks and allows for concurrent execution of multiple subtasks by agents. It also reduces redundancies by enabling the agents to identify the exact requirements of each task. The task CPN is not embedded into the agents and is dynamically selected by the task managers. Thus it is not limited to the specific task CPN model used for the experiments. New task CPN models can be defined for various types of tasks and domains, and the agents are equipped with a mechanism that allows them to select the appropriate task model according to the requirements of specific tasks.

The framework was empirically tested by employing the CPN formalism for tasks and roles. The results demonstrated how the framework can be outfitted with straightforward learning mechanisms in order to enhance agent team formation and task execution in decentralised multi-agent environments. We have demonstrated that when the number of tasks was high, the preferred strategy for the circumstances presented was the Nearest available, which allowed the agents to perform an increased number of tasks. However, when the task density was low the preferred strategy for the circumstances presented was Best available, which allowed the agents to perform a larger portions of tasks. The effect of adaptation to unreliable agents was also shown. The adaptive mechanism that was proposed for the task manager role enables the agents to adapt to unreliable circumstances. This is a useful mechanism that allows the task managers to replace the agents that have not fulfilled their commitments (i.e.,
have not performed the subtask they had committed to perform).
Chapter 7

Robotic Experiment

The framework proposed in chapter 6 was also tested using physical robots. Since the framework was designed for a distributed system, we were able to transfer the framework to physical robots with only minor changes. The physical deployment of the framework ensured that the framework met the requirements of a physically distributed and concurrent system.

The consideration of realistic features, including concurrency and distributed design enabled us to directly transfer our developed framework to physical robots. This is a key difference between our work and other work mentioned in the literature, since most simulation approaches cannot be easily transferred to physical systems due to unrealistic assumptions.

This chapter introduces the technologies used for the physical experiment. It introduces the Garcia robot and the technologies used for the experiment. In addition, it provides the details of the physical environment that we set up in order to test the framework.

7.1 Garcia robot

The Garcia is a physical robot that features an on-board computational host (Acroname, 2005a). Figure 7.1 shows a picture of the Garcia robot. Without a host the robot can perform simple processes such as motion control and running simple programs. However, in order to perform more complicated behaviours, it requires a processing host. The host can use the Garcia API to directly communicate with the Garcia’s internals. The Garcia API is a software object that manages threads for robot control and communication as well as monitoring
the status of the link between the robot and the API. The Garcia API provides routines for creating behaviours, assigning behaviour parameters, scheduling behaviours, and checking the results of each robot action. The API handles queues of behaviours (or tasks) that the robot is to perform. These behaviours are executed one at a time and in the order that they are received. The Garcia API constantly monitors the robot’s status by checking for the following:

- Obstacles in view of the front range finders
- Obstacles in view of the rear range finders
- Loss of direction from downward proximity sensors
- Stall on one or both motors
- Reception of an infra red command

7.1.1 Garcia host

The processing host represents the brain of the robot. The following shows the basic categories for the host configurations

- Remote host
- Local host on the robot
- Host network
• No host

In the remote host configuration, the robot is linked to a desktop computer via the serial port or wireless connection. The robots can also carry their own local host that can be a Stargate ¹, Gumstix ² or a Personal Digital Assistant (PDA). We used Stargate and Gumstix local hosts for our experiments. The host network configuration takes a host on-board to allow a remote robot to remotely control the current robot. The no host configuration limits the capability of the robot, but it can still perform useful tasks, although it requires the user to have an in-depth knowledge of the hardware and operating system.

**Stargate**

Stargate is a small computer which features an Intel 400 MHz Xscale processor to provide processing capability on small devices (Acroname, 2005b). The operating system is based on Linux. It features a daughter card which provides a Flash type III slot. This slot was used to connect the wireless LAN card. In addition it has Ethernet, USB and RS-323 serial ports. The serial port was used to communicate with the Stargate board. The Stargate boards run Linux 1.6.

**Gumstix**

Gumstix currently has two products: the Texas Instruments OMAP-based Overo series and the Marvell XScale-based Verdex Pro series. Our robots use the Gumstix Verdex Pro series (Acroname, 2008). The Gumstix products offer functions including Bluetooth and 802.11g wireless interfaces, synchronous and asynchronous serial, USB, Ethernet, and RS-232 in a compact form. The operating system of the Gumstix is OpenEmbedded Linux.

### 7.1.2 Primitives

Primitives are the basic building blocks of the Garcia robot’s behaviours. An example primitive is traversing a specified distance or turning with a specified angle. The Garcia primitives are the essential components of motion and action from which high-level operations can be

constructed. The primitives are implemented in the form of plug-in modules. Each primitive has a set of properties that control its operation. Once a primitive is associated with specific values for its properties, it becomes a behaviour.

### 7.2 RFID technology

RFID (Radio Frequency IDentification) refers to electronic devices that consist of tags, antennas and readers. There are two types of tags: active and passive. Active tags have their own power source and thus can be read from a longer distance, whereas the passive tags can only be read from a short distance. An RFID reader is used to read and write information from and into a tag. The RFID reader typically has an antenna which creates short-range radio frequency signals. The radio frequency signals feed the passive RFID tags with the required energy so they can communicate with the reader.

![Passive RFID tag](image1.png) ![RFID reader](image2.png)

Figure 7.2: Passive RFID tag. Figure 7.3: RFID reader.

This work used ISO18000-6C RFID tags and M9 modules which are Ultra High Frequency (UHF) RFID readers. The ISO18000-6C tags allow the users to write up to 6 bytes of data into the tags. Figures 7.2 and 7.3 show the pictures of the RFID tags and M9 module (RFID reader), respectively. Figure 7.4 shows the Skyetek UHF antenna used for our experiments. Each RFID tag in our experiments contains the coordinates of the region in which it is residing in addition to the task information.
7.3 Description of the physical agent platform

Each Garcia robot was set to run an Opal agent. The Opal agent used its CM module and was therefore able to interact with multiple robots concurrently. All the robots were WiFi enabled, allowing a local network to be formed among robots. Ambicom WL5400G-CF wireless cards (Ambicom, 2007) were used for the robots with Stargate hosts, and the robots with Gumstix boards had built in WiFi cards. The robots were able to communicate with each other by sending messages and using the Hypertext Transfer Protocol (HTTP).

Each robot was equipped with an M9 RFID reader and a UHF broadband antenna. The readers were connected to their hosts via the USB port. This equipment enables the robots to read the tag’s information as the robots move within the environment to determine their location. Figures 7.5 and 7.6 show pictures of a robot that is equipped with the RFID reader and antenna. The tags’ location information enabled the robots to plan their path to their destinations when they were on the grid.
7.4 Experimental setup

Acroname provides a Java API for the Stargate. However, the robots that use the Gumstix board only have a C++ API. Therefore we developed some Java Native Interface (JNI) code in order to connect our Java-based agents to the robot API on Gumstix. In addition, a JNI code module was developed for the RFID API which was written in the C language. Finally a shared library was created for Linux so that the JNI code could access the RFID and robot APIs provided in the C and C++ languages.
7.5 Experimental result

We created a grid environment to test the physical instance of our framework. A grid of 5 by 5 was created and a RFID tag was deployed in each cell of the grid, where each cell represents a region. Each tag had the coordinates information for its corresponding region. In addition to location information, each tag contained the task information when there was a task in that region.

The first two bytes of each RFID tag (ISO18000-6C) contains the geographical coordinates of the location (cell) and the third byte represents whether there is a task in that location (it contains 1 where there is a task in the area otherwise, it contains 0). The fourth and fifth bytes represent the task requirements. In our example task, requirements are defined using colours: W for a white robot and Y for a yellow robot.

Figure 7.7 shows the experimental grid and the agents’ locations at the beginning of the experiment. Figure 7.8 shows how the agents communicated by sending and receiving message tokens from the CPN models. Note that details of this communication had been described in term of agent’s role in chapter 6. Figure 7.9 shows a more abstract representation of communication among robotic agents. The information received through communication between agents, enables an agent to decide what action to take. Once the action is decided,
the agent sends a signal to the robot’s actuators which leads to a robot action such as moving straight, turn left or right, and spinning around its axis.

![Communication among agents.](image)

Figure 7.8: Communication among agents.

![Schematic representation of communication among agents.](image)

Figure 7.9: Schematic representation of communication among agents.

We put a virtual task into one region by writing the task information into the RFID tag of that region. For our experiment, four robots were positioned in various edges of the grid. The robots had different colours representing their resources. Each robot was equipped with a RFID reader and an antenna. As the robot moves, the RFID reader reads the tag information. One robot (the red robot) was assigned to be the initiator robot and was able to detect the tasks while the other robots were assumed to be the helper robots. At first, the initiator moved randomly in the environment until it detected the task. Once a task was detected, the initiator changed its role to the task manager and started looking for teammates. To achieve
this, the task manager sent a request to all the helper agents. Once it received the responses, it started processing the requests and forming the team. It was assumed that the task requires one white and one yellow robot. Once all the team members moved to the task location, they started performing the task. The task CPN required two robots to work concurrently. For this example task, the white and yellow robots must initially perform their tasks concurrently. The robots spun around on their axis to simulate performing their tasks. Once the two robots performed their tasks, they sent a message to the red robot informing it that the task has been performed. The red robot then spun around its axis to indicate that it has received the messages from all the teammates, and the task was completed.

7.6 Summary

This chapter has provided the details of our elementary physical instantiation of our cooperation framework. The framework was validated and tested by a group of three Garcia robots that were used to perform a complex task. The simulation environment was created by using RFID tags. The RFID tags stored the location information in addition to any task information. Each robot was equipped with a RFID reader that allowed the agents to localise themselves within the grid environment in addition to reading the task information. Each robot was running an Opal agent. A local network was formed among the agents. The WiFi network technology network enabled the agents to communicate. The Opal agents communicated using FIPA ACL (FIPA, 2002a) messages and coordinated their activities by passing asynchronous messages to each other. These messages were processed by the CM of each agent. The CM directed each message to the appropriate CPN by using its conversation ID for identifying the associated CPN model. The Garcia API was used to send commands to the robots.

Advanced experiments that require the physical robots to run for long periods of time cannot be performed using the current set up. That is due to the fact that the RFID readers and robots consume the robot’s batteries relatively quickly. However, this chapter demonstrated the nature of how the developed framework can be deployed on physical robots. Our FIPA-compliant framework is a step towards real practical agent systems where agents may cooperate and coordinate their activities for performing complex tasks.
Chapter 8

Discussion

The first part of this chapter discusses the contributions of this thesis. In the second part, the limitations and possible future directions of the framework are described, and the last part concludes our work.

8.1 Contribution of this thesis

The contribution of this thesis is in the area of cooperation and coordination in multi-agent systems. It has focused mainly on the concurrent aspects of agents cooperation to reflect real-life scenarios where agents act in a concurrent fashion in order to solve complex problems. The source code for this work is written mainly in Java and can be made available upon request.

8.1.1 Modelling agents’ concurrent activities

The framework uses a multi-threaded agent approach in which each agent runs in its own thread and consequently each agent runs its own life cycle, and is able to perform reactive and proactive behaviours. Chapter 4 of the thesis focused on introducing the basic components of the framework. The framework initially used primitive agents and later replaced them with autonomous agents. Autonomous agents created a multi-threaded framework that allows for modelling realistic situations in which agents interact with each other concurrently. Such considerations allowed agents to interact in a more realistic concurrent environment.
Various strategies were designed for the agents. The strategies enable the agents to select appropriate teammates according to the agents’ attitudes. In addition, the agents can alter their strategies depending on the various constraints of the environment. Thus the strategies allow for flexibility and adaptability of the agents of the framework under different conditions.

Chapter 5 described how Opal agents, which are advanced autonomous agents, were employed. The framework employed Coloured Petri Nets to model agents’ concurrent behaviours. Petri Nets are a formalism that has an associated graphical representation and are used for modelling dynamic and concurrent systems. Coloured Petri Nets are suitable for representing concurrent activities of agents, for instance, two transitions may be enabled concurrently (if tokens for enabling both transitions are present) thus allowing for concurrent execution of two actions. In our multi-threaded agent framework all the agents run concurrently (each agent has its own thread), and each agent is enabled to create a separate conversation (Coloured Petri Net model) for interacting with other agents. Therefore, the framework mimics real-life concurrent scenarios where an entity may interact with several other entities and it is able to keep track of its various conversations. Each agent may play a different role in each conversation, and thus an agent may play several distinct roles in separate conversations concurrently. As a consequence, the developed framework supports a high degree of concurrency. Chapter 5 of this thesis showed how CPNs were used to model agents’ conversations.

Opal agents use an asynchronous non-blocking method for communication among agents. The asynchronous method improved autonomy of agents by enabling them to operate after sending a message without waiting indefinitely for their response. The framework is capable of dealing with uncertainties associated with message delivery by adding a ‘timeout’ mechanism to the Petri Net model of the roles. The ‘timeout’ specifies the maximum time that an agent is allowed to wait in situations where the agent expects to receive responses. An agent resumes its process if it receives all the responses from the requesting agents, or the waiting time elapses.
8.1.2 Modelling tasks

The framework also uses Coloured Petri Net for modelling its tasks. Each task CPN specifies the interdependencies among its subtasks. It also enables concurrent execution of multiple subtasks. In a Petri Net, various transitions can be fired concurrently if there is a token in the input 'place' of the transitions. The task CPN defines ‘assign requirements’ transition that assigns various subtasks (task requirements) to different agents. Therefore, multiple Petri Net ‘places’ in a task Petri Net (places that contain the ‘subtask’ tokens) may receive their tokens at the same time. This results in the concurrent execution of multiple transitions.

The agents can use different CPN models for tasks according to the requirements of each tasks. The task CPN is updated and managed by a task manager agent that executes the task CPN by employing the JFern CPN execution engine (Nowostawski, 2000) based on the information that the task manager receives from the participating agents. Chapter 6 introduces the task CPN and how they were used to model tasks and monitor their execution. The use of CPNs in the design and execution stages of the software lifecycle supports a more robust software engineering approach to situated agent-based software engineering.

8.1.3 Adaptation

The framework developed and presented here enables the robotics agents to adapt under dynamic circumstances. We simulated an open and dynamic environment where new agents can join and current agents may leave the system. In addition, new tasks are generated and old tasks are removed, thus creating a highly dynamic environment. We showed how helper agents can change their roles to that of task manager when required. Thus the agents can change their roles at run-time if required. Similarly, agents can also employ different CPN models for tasks depending on the task requirements (although we have not shown any experiment that demonstrates the use of different task models). In addition, agents can change their strategies under dynamic situations by using a computationally straightforward and efficient reinforcement learning technique (Sutton and Barto, 1998). The adapting mechanism allows agents to find the most appropriate strategy to work with under various circumstances. We also showed how agents can identify reliable agents and avoid interacting with unreliable agents under certain circumstances. Agents in these situations used a reinforcement learning
technique to update their reliability tables. Chapter 6 includes the result of our adaptation mechanism. The adaptation scenarios were designed in a way that the effect of adaptation can be clearly demonstrated.

### 8.2 Future research directions

In this section we describe the future research directions that could build on the work presented in this thesis.

1. The current version of the framework does not support a hierarchy of tasks, for example, a subtask of a complex task can be a complex task itself which requires coordination and cooperation of multiple agents. Since the JFern framework supports hierarchy, that feature could be added to the system in the future to support more complex scenarios.

2. The Framework uses a greedy approach for task allocation. In the simulation framework, the use of the synchronisation technique does not allow concurrent multiple access to a task object. However, in the physical experiment there may be situations where a task is identified by multiple agents concurrently. In such situations, several agents may start the task allocation process concurrently. Therefore there would be added value in having an additional mechanism that only allows one agent to allocate a task. One possible solution could be for the agents that identify the tasks to broadcast their tasks to all other agents in the same area (in the case of our example scenarios this refers to a broadcast message to all other initiators in the region). The agent that has the least distance to the task then would get to allocate the task. If there are several agents with the same distance, then one of the agents is determined (either randomly or according to “first come, first served”), and then informs the other agents about the decision.

3. The framework has the capability for allowing agents to work on multiple tasks. Future example scenario could study a situation in which a helper agent may accept more than one request. In these situations, each helper agent has a queue of all the tasks that it
has committed to perform and the estimated time that it will become available for each of the tasks. The helper agents may then inform the task managers of the estimated time for their availability. In addition, if more urgent tasks appear in the meantime, the helper agent may reschedule its commitment queue and inform the task managers about the change in its time of availability. Such situations are especially important where some of the resources are scarce and an appropriate allocation of resources is critical.

4. Another example scenario that could further demonstrate the capabilities of the framework, is where there are several problems that may belong to different domains. The framework allows the definition of multiple Interaction Protocols (IPs) where each IP refers to a given problem domain. Under each IP the developers may specify several roles that may be required in performing tasks of such domains, for instance, an agent that is performing in a rescue situation may face a hazardous situation due to the unforeseen presence of chemicals in an area. The framework allows agents to change their IPs depending on the problem domain and start forming teams and executing tasks based on the requirements of each individual task.

5. A token-based mechanism could be used to perform tasks similar to the work done by Scerri et al. (2005). In order to do that, an agent that finds a task passes a copy of the task CPN in the form of a token to an agent in its neighbourhood. Each agent needs to check the task requirements and its capabilities before it can commit to a task. If an agent does not have the required capability, then it would pass the task to another agent in the region. If an agent does accept a potential token, it commits to perform the task once the interdependencies have been worked out. The number of agents that the token can be passed to must be limited so that the token is not transferred to too many agents. This approach would basically eliminate the need for the task manager.

6. Petri Nets offer tools for verification and analysis of the models. We have partially verified our CPN models by ensuring that certain places have been reached (reachability). However, more detailed and formal reachability analysis can be performed (Murata, 1989; Jensen and Kristensen, 2009).
8.3 Conclusion

We developed a novel software engineering framework that uses AOSE concepts and techniques to facilitate coordination of agents in situations where agents need to form teams and cooperatively perform tasks. The framework ensures reliable deployment of our coordination schemes and mechanisms in real situations. The framework, which supports the design and test of cooperative agent-based systems and then the direct deployment of the designed system onto the instantiated physical systems, goes beyond existing work in the area.

The framework presented here is a generic cooperation framework that can be used in various cooperative domains. It can be used for coordinating and managing the development and operation of a system of robotic agents that operate in a distributed spatial environment. It also facilitates the formation of agent teams, without centralized control, in the context of spatial and temporal constraints. Each agent in the framework represents a robot; thus the agents move from one location to another in order to find and perform tasks. Agents have limited local visibility range, so they do not have complete information about other agents and the environment. In addition, the framework enables the agents to use several teammate selection strategies which affect the decision-making of the agents for selecting teammates. The strategies were designed as a separate module and they can be easily replaced by any other user-defined strategies. The designed strategies reflect the effect of distance, time and agents capabilities. The aim of this thesis was not to identify the universal best strategy, but we are aiming to find practical and distributed agent-based solutions that can be easily mapped to physical robots in order to facilitate coordinated problem solving among robotics agents. The result of the experiments greatly change with the change of parameters, such as the number of agents, visibility ranges, and waiting times for receiving responses. In fact the strategies were designed to demonstrate the various dimensions of the framework.

The framework provides a semi-concurrent simulation framework by employing FIPA-compliant multi-threaded agents. The multi-threading feature enables us to simulate real-world scenarios in which several agents run and interact with each other concurrently. It also uses FIPA-ACL messages for agent communications. Thus the FIPA-compliant feature of the framework provides interoperability with other FIPA-compliant agent platforms.

In order to support concurrency, the agents’ interactions were modelled by using the
Coloured Petri Net modelling technique. CPNs offer concurrent modelling at a high abstraction level which provides formal models with mathematical formalism. CPNs are well suited for simulating, analysing and modelling distributed and concurrent systems. The CPNs support precise modelling and analysis of system behaviour, while the graphical representation of CPNs facilitates intuitive understanding of the proposed solution. The CPN model enables the agents to keep track of their conversations with various agents and resume a conversation when a message arrives.

The CPN technique was also used to model the tasks. Each task has several requirements, and each requirement can be performed by a single or several agents. A task manager is assigned for each task. The task manager is in charge of assigning task requirements to agents. The use of task CPNs facilitates agent management in terms of the order in which various requirements must be performed. This provides a good scheme for specifying task requirements, since it shows the order of subtasks and their interdependencies. This feature is especially useful when different skills are required in different phases of the task. Without such a facility, agents may erroneously recruit extra resources that may not be required.

The use of (a) threads for each agent, (b) the CPN modelling technique, and (c) asynchronous communication among agents resulted in a fully distributed system that is scalable and can be applied in large-scale distributed systems without any implicit constraint on the size of the system.

The framework can be used for run-time adaptation. The agents can change their roles at run-time, where each role specifies the CPN model that each agent must execute for each conversations (conversation model). The framework has the potential for adapting to different problem domains, for example, each agent can have several IPs and their associated role models, therefore, enabling the agents to select the appropriate CPN model based on the IP and role information specified in the body of each message (this potential change of the IP was not empirically demonstrated in this thesis). The change of roles was covered in chapter 6, where agents playing the helper role change their role to that of task manager. Moreover, various task CPNs can be used for different types of tasks. Therefore, users can deploy the framework in different domains by plugging in their application-specific CPN models for roles and tasks in the framework.

This thesis has used reinforcement learning (Sutton and Barto, 1998) for agents learning
and adaptation. Agents can change their team building strategies dynamically, depending on the various constraints encountered in the environment. Our experiments showed that when the number of tasks was high the *Nearest available* strategy, which allows the agents to perform a greater number of tasks by selecting the closest agents, was the preferred strategy for the circumstances under test. When the number of tasks was low, the preferred strategy was the *Best available* strategy that allows the agents to perform larger portions of tasks for the circumstances under test. In addition, we have shown how agents can adapt to situations where agents are not reliable. Our learning mechanism allows agents to identify the unreliable agents and avoid interacting with them if more reliable agents are available.

The use of distributed agent-based techniques enabled us to transfer the framework to physical robots with only minor changes. The physical experiment using Garcia robots demonstrated the efficiency and effectiveness of our framework in physically distributed scenarios. Our physical experiment can be viewed as a step towards real application of agents in physically distributed and concurrent systems where agents autonomously interact and coordinate their activities to solve complex problems. Such approaches can lead to dramatic improvements in the scope and efficiency of problem solving for real, situated physical agents involved in complex and distributed tasks.
Appendix A

An example of interaction among agents using the conversation manager

In order to understand how the agents execute the Coloured Petri Net models of their roles, we provide an example. In this example two agents, start a conversation. The initiator of the conversation uses the model showed in Figure A.1, and the participant uses the model described in Figure A.4. The initiator initialises its CPN model with the name of the agents in its neighbourhood (agents in its visibility range) and sends a request to those agents. Figure A.2 shows the Java code associated with the send request transition, where ‘x’ refers to the name of the variable associated with the arc that connects the start place to the transition send request. Variable ‘z’ is associated with the arc that connects the neighbouring agents place to the send request transition. ‘z’ stores the list of the agents in the neighbourhood. The initiator sends a FIPA ACL message with PROPOSE performative to all the agents in the list. If there is no agent in the neighbourhood, then the agent removes the current conversation from the list of conversation. This is done by calling the addToListOfOldConversations method. It also calls the undoTeammingCommitment method that changes the agent’s current status to ‘not busy’, so the agent can start moving and looking for new tasks. The call method provides a means for calling methods on the agent itself from CPN model.
Figure A.1: The CPN model of the initiator.

```java
String content = get("x").toString();
((CpnContext)getCpnContext()).remVar("x");
List nameList = (ArrayList) get("z");
convID = content.substring(content.lastIndexOf("=")+1+".length().replace(",","\"\").trim();
if (nameList.size() > 0) {
    Message newMsg = new Message();
    newMsg.set(Message.CONTENT, content);
    newMsg.set(Message.PROTOCOL, "robotcooperation");
    newMsg.set(Message.LANGUAGE, "FIPA-SL");
    newMsg.set(Message.ONTOLGY, "robotcooperation");
    newMsg.set(Message.CONVERSATION_ID, convID);
    newMsg.set(Message.ACT, Message.PROPOSE);
    String name = nameList.get(0).toString();
    newMsg.set(Message.RECEIVER, new BasicAgentName(name));
    ((CpnContext)getCpnContext()).getVarPool().put(name, newMsg);
    nameList.remove(0);
} else {
    call("RobotCooperationInitiatorMethod","addToListOfOldConversations",new Class[] {String.class, String.class}, new Object[] {myName, convID});
    call("RobotCooperationInitiatorMethod","undoTeamingCommitment",new Class[] {String.class}, new Object[] {myName});
    ((CpnContext)getCpnContext()).getVarPool().put("end","end");
}
((CpnContext)getCpnContext()).getVarPool().put("convID", convID);
((CpnContext)getCpnContext()).getVarPool().put("neighbors", nameList);
((CpnContext)getCpnContext()).remVar("z");
```

Figure A.2: The code corresponding to the send request transition.
Figure A.3 shows the CPN model of the participating agents. Figure A.4 shows the Java code associated with firing the send response transition. Once an agent receives a message, it checks the performative parameter of the FIPA message. Depending on the performative, the agent decides what to do. If the message performative from the other agent is ‘proposal’, the receiving agent checks its availability by calling the checkForAvailability method. This method returns a boolean value that refers to the availability of the agent. If the agent is available then the send response transition sets the message performative to ‘accept’, otherwise it sets it to ‘reject’. An agent accepts a request if it is not committed to another task. If the agent decides to accept the request, then the call method calls the changeMyStatusToBusy method on the agent, which subsequently changes the status of the the agent to busy, meaning that the agent cannot commit to any other contract.
Message m=(Message) get("x");
((CpnContext)getCpnContext()).remVar("x");
Object validate=
call("RobotCooperationParticipantMethod","checkForAvailability",new Class[]
{String.class,new Object[] {myName} } );
boolean available = ((Boolean)validate).booleanValue();
String senderName=m.get(Message.SENDER).toString();
Message msg = new Message();
msg.set(Message.CONTENT,m.get(Message.CONTENT).toString());
msg.set(Message.RECEIVER, new
BasicAgentName(m.get(Message.SENDER).toString()));
msg.set(Message.LANGUAGE, "FIPA");
msg.set(Message.ONTOSTY, "robotcooperation");
msg.set(Message.PROTOCOL,m.get(Message.PROTOCOL));
String receiverName=m.get(Message.SENDER).toString();
if(available){
    call("RobotCooperationParticipantMethod","changeMyStatusToBusy",new
Class[] {String.class,String.class},new Object[] {myName,receiverName} );
    msg.set(Message.ACT, Message.AGREE);
    ((CpnContext)getCpnContext()).getVarPool().put(rereceiverName,msg);
    ((CpnContext)getCpnContext()).getVarPool().put("accept","accept");
} else {
    msg.set(Message.ACT, Message.REJUSE);
    call("RobotCooperationParticipantMethod","addToListOfOldConversations", new
Class[] {String.class,String.class},new Object[] {myName,conversation_id} );
    ((CpnContext)getCpnContext()).getVarPool().put(receiverName,msg);
}

Figure A.4: The code corresponding to the send response transition.
References


CPN group at the University of Aarhus (2001). CPN Tools.


167


173


