University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Self-management Framework for Mobile Autonomous Systems

Eskindir Ayallew Asmare

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, March 2011

Abstract

The advent of mobile and ubiquitous systems has enabled the development of autonomous systems such as wireless-sensors for environmental data collection and teams of collaborating Unmanned Autonomous Vehicles (UAVs) used in missions unsuitable for humans. However, with these range of new application domains comes a new challenge – enabling self-management in mobile autonomous systems. The primary challenge in using autonomous systems for real-life missions is shifting the burden of management from humans to these systems themselves without loss of the ability to adapt to failures, changes in context, and changing user requirements. Autonomous systems have to be able to manage themselves individually as well as to form self-managing teams that are able to recover or adapt to failures, protect themselves from attacks and optimise performance.

This thesis proposes a novel distributed policy-based framework that enables autonomous systems to perform self-management individually and as a team. The framework allows missions to be specified in terms of roles in an adaptable and reusable way, enables dynamic and secure team formation with a utility-based approach for optimal role assignment, caters for communication link maintenance among team members and recovery from failure. Adaptive management is achieved by employing an architecture that uses policy-based techniques to allow dynamic modification of the management strategy relating to resources, role behaviour, team and communications management, without reloading the basic software within the system.

Evaluation of the framework shows that it is scalable with respect to the number of roles, and consequently the number of autonomous systems participating in the mission. It is also shown to be optimal with respect to role assignments, and robust to intermittent communication link disconnections and permanent team-member failures. The prototype implementation was tested on mobile robots as a proof-of-concept demonstration.

Acknowledgements

I am deeply grateful to my supervisor, Professor Morris Sloman, for his constant advice, guidance and support throughout the years of my PhD study. This thesis would not have been possible without his encouragement, constructive criticism and close attentiveness in reviewing my work.

I am very grateful to my second supervisor, Dr. Naranker Dulay, for his guidance and critical reviews, which helped improve the quality of this thesis. I am also very grateful to Dr. Emil Lupu for his helpful comments and advice. Special thanks go to Professor Keith Clark for his invaluable advice and comments.

I would like to thank the academic staff and researchers in our department especially, Dr. Kevin Twidle, Dr. Alessandra Russo, Dr. Sye Loong Keoh and Dr. Arosha Bandara for their help and advice.

Special thanks also go to my colleague in the Self-managed Mobile Cells project, Dr. Anandha Gopalan, who has contributed a great deal to the work presented in this thesis with his useful comments.

Many thanks to fellow students and friends – Alberto Schaeffer-Filho, Lucio Duarte, Markus Huebscher, Paulo Maia, Daniel Sykes, Changyu Dong, Leonardo Mostarda, Giovanni Russello, Dalal Alrajeh, Driss Choujaa, Robert Craven, Yanmin Zhu, Vrizlynn Thing, Duc Le, Andrew Smith, Srdjan Marinovic, Enrico Scalavino, Domenico Corapi, William Heaven, Rudi Ball, Dimosthensis Pediaditakis, Themistoklis Bourdenas and Jiefei Ma.

Many thanks also go to Atnafe and Ashagre, and to my friends – Bizen, Fetahi, Yosef and Belachew for their support and encouragement throughout the years of my study.

I am deeply grateful to my parents, Genet Gebreab and Ayallew Asmare, to whom this thesis is dedicated, my sisters – Yodit and Eleni, and my brothers – Kenya, Biniam and Ermias for their constant support and encouragement.

The thesis was funded by the Systems Engineering for Autonomous Systems (SEAS) Defence Technology Centre (DTC) established by the UK Ministry of Defence, and the Overseas Research Students Awards Scheme.

Statement of Contribution

This thesis is the result of the author's work at Imperial College London on policy-based management of distributed systems. Many of the ideas developed in this thesis are the result of group discussions with Professor Morris Sloman, Dr. Naranker Dulay, Dr. Emil Lupu and Dr. Anandha Gopalan. The thesis builds upon previous work within the Department of Computing on policy-based management and the self-managed cell concept.

The design and implementation of the policy-based mission and team management layers of the framework are the author's individual work while the design of the communication link maintenance element of the communication management layer is a collaborative work with Dr. Anandha Gopalan. The communication link maintenance element is implemented by Dr. Anandha Gopalan.

List of Publications

Eskindir Asmare, Anandha Gopalan, Morris Sloman, Naranker Dulay, Emil Lupu. **Communication and Failure Management Schemes for the Self-Management Framework of UXVs.** Systems Engineering for Autonomous Systems Defence Technology Centre Conference, July 2009, Edinburgh, UK.

Eskindir Asmare, Anandha Gopalan, Morris Sloman, Naranker Dulay, Emil Lupu. **A Mission Management Framework for Unmanned Autonomous Vehicles.** In Proceedings of the Second International ICST Conference on MOBILe Wireless MiddleWARE (Mobilware 2009) , Operating Systems, and Applications, April 2009, Berlin, Germany.

Eskindir Asmare, Anandha Gopalan, Morris Sloman, Naranker Dulay, Emil Lupu. **A Policy-Based Management Architecture for Mobile Collaborative Teams.** In Proceedings of the Seventh Annual IEEE International Conference on Pervasive Computing and Communications (PerCom 2009), March 2009, Galveston, TX, USA.

Eskindir Asmare, Anandha Gopalan, Morris Sloman, Naranker Dulay, Emil Lupu. **Adaptive Self-management of Teams of Autonomous vehicles.** In Proceedings of the 6th International Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC 2008), December 2008, Leuven, Belgium.

Eskindir Asmare, Naranker Dulay, Anandha Gopalan, Emil Lupu, Morris Sloman. **Secure Distributed Self Management Framework for UXVs.** Systems Engineering for Autonomous Systems Defence Technology Centre Conference, June 2008, Edinburgh, UK.

Eskindir Asmare, Naranker Dulay, Emil Lupu, Morris Sloman, Seraphin Calo, Jorge Lobo. **Secure Dynamic Community Establishment in Coalitions.** In Proceedings of IEEE Military Communications Conference (MILCOM 2007), Orlando, FL, USA.

Eskindir Asmare, Morris Sloman. **Self-management Framework for Unmanned Autonomous Vehicles.** In Proceedings of Inter-Domain Management, First International Conference on Autonomous Infrastructure, Management and Security, Oslo, 2007, Springer Berlin / Heidelberg.

Eskindir Asmare, Naranker Dulay, Emil Lupu, Morris Sloman. **Towards Self-managing Unmanned Autonomous Vehicles.** Systems Engineering for Autonomous Systems Defence Technology Centre Conference, July 2007, Edinburgh, UK.

Eskindir Asmare, Naranker Dulay, Hahnsang Kim, Emil Lupu, Morris Sloman. **Management Architecture and Mission Specification for Unmanned Autonomous Vehicles.** Systems Engineering for Autonomous Systems Defence Technology Centre Conference, July 2006, Edinburgh, UK.

Dedication

ለወላጆቼ
ገነት ገብረአብ እና አያሌው አስማረ

To my parents
Genet Gebreab and Ayallew Asmare

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AI Artificial Intelligence

ANS Agent Naming Service

ANS Autonomic Nervous System

CDL Configuration Description Language

CIM The Common Information Model

DAML DARPA Agent Markup Language

DCSP Distributed Constraint Satisfaction Problem

DMTF Distributed Management Task Force

DOTN Distributed Organisational Task Network

IETF Internet Engineering Task Force

ITL Information Terminological Language

KQML Knowledge Query and Manipulation Language

LPG Local search for Planning Graphs

MANET Mobile Ad hoc Network

MILP Mixed Integer Linear Programming

MOF Managed Object Format

NHC Nested Hierarchical Controller

OGSA Open Grid Service Architecture

PCIM Policy Core Information Model

PDDL Planning Domain Definition Language

PDP Policy Decision Point

PEP Policy Enforcement Point

PMA Policy Management Agent

PSTC Provisioning Services Technical Committee

RCS Real-time Control System

SDP Session Description Protocol

SMC Self-managed Cell

SPML Service Provisioning Markup Language

TR Teleo-reactive

UAV Unmanned Autonomous Vehicle

UPnP Universal Plug and Play

WSDL Web Services Description Language

Chapter 1

Introduction

The advent of mobile and ubiquitous systems has enabled autonomous systems ranging from tiny wireless-sensors used for environmental data collection to wearable computers used for monitoring a patient's health to Unmanned Autonomous Vehicles (UAVs) used in missions unsuitable for humans. However, with these range of new application domains come a new challenge – *enabling self-management in mobile autonomous systems*. For example, collaborating teams of UAVs have been deployed in space exploration, search and rescue and other similar missions where involving humans is costly or infeasible. In most of these missions, it is infeasible for humans to be in the control loop. This necessitates that management tasks be shifted to the managed autonomous systems.

Managed autonomous systems should be able to configure themselves automatically in accordance with high-level mission specifications and need to adapt their behaviour to current context such as location and activity, available resources such as battery power and available services such as (intermittent) wireless communication links. They should be self-managing in that they have to self-configure, recover or adapt to failures, protect themselves from attacks and optimise performance to best utilise available resources. This thesis attempts to address the issues of self-management in mobile autonomous systems in general and UAVs in particular.

1.1 Mobile Autonomous Systems

The term *autonomous system* is used to describe various notions of autonomy in different areas of computer science and engineering. In *Computer Networks*, an autonomous system is a set of routers under a single administrative domain that presents a consistent picture to others of what destinations are reachable through it [RLH06]. In *Robotics*, an autonomous system (robot) is one that is capable of operating in the real-world environment without external control for an extended period of time [Bek05]. However, in some application domains such as space exploration, autonomous systems are automated [THRR06] but not necessarily self-managing. Our definition of an autonomous system incorporates both the ability to operate without external control and the ability to perform self-management. We define an autonomous system as one that is capable of accomplishing its mission without human intervention by managing its resources and behaviours using its self-management (autonomic) capabilities. Mobile autonomous systems are autonomous systems that have the ability to move. A typical example is a mobile robot. UAVs are one type of mobile robot that can be aerial, (over) underground or (over) underwater. Generally UAVs include some if not all functionalities such as overland movement and propulsion, autonomous navigation, adaptive mission planning, sensing (e.g., infra red, explosive material detection, radiation or chemical detection), target recognition or identification, and situation assessment. They also have processing, communication, and data storage capabilities. UAVs can be tele-operated or capable of autonomous navigation.

UAVs are being used increasingly in civilian and military missions that are dangerous or otherwise impossible for humans. Some examples are shown below.

- On September 11, 2001, the Center for Robot-Assisted Search and Rescue (CRASAR) used tele-operated-robots in a mission that was the first of its kind, for urban search and rescue [Mur04b]. The robots were used for searching for victims, paths, structural inspection and detection of hazardous materials. In 2005, small unmanned aerial vehicles were used during Hurricane Katrina to search for survivors and assess conditions along Katrina's path through Mississippi and larger vehicles were used during Hurricanes Dennis and Rita [MPB08]. In the same year, unmanned water-surface vehicles and smaller aerial vehi-

cles were used in the aftermath of Hurricane Wilma for post-disaster inspection [MSG+08].

- In 1991, *Terregator*, an unmanned ground vehicle was used to map parts of a coal mine [CBCD91]. The use of mobile robots has now expanded to other subterranean spaces such as caves and sewers [MFO+06].
- Unmanned vehicles are also being used for clearing land mines [Hab07], border patrol [Phi07] and various military applications.
- In 1970, *Lunokhod 1*, the first unmanned vehicle landed on the moon [She71]. This tele-operated-vehicle was used to collect lunar soil samples. The use of unmanned vehicles for space exploration has grown since then. In 1997, *Sojourner*, a semi-autonomous unmanned vehicle landed on Mars and navigated to many sites while conducting experiments using its onboard devices [WN98]. In January 2004, two unmanned vehicles, *the Mars Exploration Rovers*, landed on Mars as part of a long-term effort to study past water activities on Mars [Jet].

1.2 Motivation

Self-managing autonomous systems, such as mobile robots, have the potential for providing the information needed to assist rescue operations by locating survivors, identifying affected areas, and helping the collaborative efforts of the response team members. They also find uses in disaster management, including earthquakes, forest fires and floods, and military applications. The primary challenge in using autonomous systems for real-life missions is specifying the missions in an adaptable way. There has been extensive research on robot control architectures, which are concerned with organising the robot primitives, i.e., sensing, planning and acting in order to enable individual intelligent robots. These architectures either do not consider mission specification as they are targeted for specific tasks [Bro86, Ark87, MA92, BAFH84, Mey93, Alb93] or their role (task) specification approach does not allow adaptation [BFG+97, Kon97, Yam04].

Merriam-Webster dictionary defines mission [Dic] as *a specific task with which a person or a group is charged*. Arkin et al. [MAC97] define mission as *a composition of tasks*. A mission, in our framework, is a specific task such as reconnaissance,

search and rescue, composed of a set of subtasks with which an autonomous system or a group of autonomous systems are charged. It is usually the case that multiple autonomous systems are deployed in a mission. Hence, the need for collaboration arises as one system can use services or resources from another system. In order to be able to form and use a dynamic collaborative team of autonomous systems in accordance with a high-level mission specification, a means for describing the capabilities of autonomous systems and discovering available heterogeneous autonomous systems is necessary. It is also imperative that the discovered systems are securely admitted to the team and assigned to an appropriate role based on their capability, and the team is maintained.

Autonomous systems are heterogeneous with respect to their hardware and control architectures. In order to check whether a given autonomous system is fit for a given task, and to compare and choose a suitable autonomous system from multiple autonomous systems, a capability description is necessary. It provides information about the existing hardware and software capabilities of the system and can be used to determine the readily available services as well as to infer the potential of the system should there be a need to dynamically enhance the system with new services.

Dynamic team formation requires a means of recruiting and vetting members. A discovery service is thus necessary to discover autonomous systems that are within the communication range of the systems that are already participating in the mission. In addition to checking the capabilities of the discovered systems against the required capabilities of the mission, it is also necessary to check the credentials of the systems before assigning them to a role in order to protect the team from malicious members. Role assignment (task allocation) may be a continuous process if there is mission replanning and consequently reconfiguration in order to improve performance or cope with contextual changes. An adaptive means of specifying team formation rules is thus necessary to allow adaptation without disrupting the operation of the team. One example system that tries to address these issues in part is CDL [MAC97], which uses a finite state machine for specifying multi-robot missions. However, the finite state machine based specification is feasible only for low-level tasks. Specifying high-level missions with many participants is not easy using this approach. It also lacks an explicit means of adaptation, dynamic role (task) assignment and security. Another example system is ALLIANCE [Par98], which deals with cooperative adaptive con-

trol where adaptation is based on local decisions made by mathematically modelled behaviours in each robot. This approach enables autonomous systems to achieve greater level of autonomy but lacks the means of enforcing organisational rules of adaptation.

It is imperative that the autonomous systems comprising the team cope with different types of failures, to function in the real-world and perform their tasks correctly. Failures can occur as a result of intermittent or permanent communication link failures as well as system failure. A study on UAV failures [CM05] shows that reliability in a field environment is only between 6 and 20 hours. In the robotic rescue mission of the September 11, 2001 disaster, a robot was lost and never recovered due to a wireless communication link failure [Mur04b]. Communication is the primary means of collaboration. It is thus important to maintain communication links among the autonomous systems in a team throughout the mission lifetime. Mobile autonomous systems mainly use wireless communication links, which can easily be affected by their movement and consequently makes communication link maintenance challenging. Two of the few documented unmanned vehicle real-life mission experiences [Mur04b, MFO+06] identify that communication link maintenance is crucial to the missions. To address this issue, previous work has either used [MFO+06] tethered systems or recommended their use [Mur04b], where the robot is connected to the commanding system by a wire that is long enough to allow the robot to move freely. However, both raised the problem associated with using communication or safety tethers, i.e., safety ropes connecting the robot to the commanding system hindered the mobility of the robot and also led it to being easily caught in the debris.

Efficient utilisation of available resources requires optimal assignment of resource users to resource providers. When forming a dynamic team, if the assignment of roles to autonomous systems is done in a manner where any capable system is immediately assigned to any available role, the team may end up in a state where more capable systems are assigned to less demanding roles. In the worst case, an autonomous system with scarce capabilities may be assigned to a role that could have been assigned to any of the autonomous systems and a role that needs the scarce capability may never be assigned thereby resulting in an incomplete team. It is thus necessary to perform optimisation on the set of discovered systems and the mission roles in order to best utilise the available resources.

The challenge in using mobile autonomous systems for real-life missions goes beyond the fundamental issues of robotics into systems management. Mobile autonomous systems need to be capable of self-management in order to cope with unpredictable real-life mission environments and attain better performance. To address this issue, a self-management framework that deals with specifying missions in an adaptable way, enables dynamic and secure team formation with optimal role assignment, maintains communication link among team members and copes with failure is necessary.

1.3 Requirements for the Self-management Framework

The self-management framework must address the fundamental tenets of autonomic computing, which are self-configuration, self-optimisation, self-protection and self-healing. It should be able to deal with heterogeneous autonomous systems and allow creation of a single autonomous entity from a set of autonomous entities. The architecture should also be scalable with respect to the number of autonomous systems participating in the mission and responsive (short response time) with respect to mission setup and adaptation.

1.3.1 Mission Management

The framework should have a mission specification scheme that enables specifying missions for teams of mobile autonomous systems. The mission specification should allow adaptation and reuse. Adaptation can be achieved by using a policy-based approach to specify the behaviours of the mission roles as well as rules for assignment of roles to autonomous systems. Reusability is necessary with respect to the mission components, for example, policies and roles, as well as the specification itself such as reusing the specification through instantiation with different mission parameters. For example, a mission specification for a search and rescue mission in a given disaster area may be reused with different instantiation parameters in another area.

1.3.2 Team Management

The framework should be able to form, use and maintain dynamic teams. It should be able to discover autonomous systems willing to join the mission, securely admit them to the team and assign them to an appropriate role using the rules of assignment provided by the mission specification. The role assignment should be optimal and the optimisation technique should have minimal memory and processing overhead in order not to adversely affect the scalability of the system with respect to the capacity of the autonomous system. The framework should also be able to detect and recover from intermittent communication link disconnection, permanent link and system failures. The detection and recovery protocols need to be policy-based so that the adaptation to failures can itself be adapted.

1.3.3 Communication Management

Although the recovery from communication link failures is considered in the team management part of the framework, prevention of failures should also be addressed. The movement of autonomous systems in a team should be controlled so that they maintain communication links and at times when this is impossible the framework should be able to respond with a measure that would make intermittent communication link available.

1.4 Assumptions

The framework is based on two main assumptions: (1) autonomous systems have a control software interface on which the management framework will act upon, (2) autonomous systems are equipped with ad hoc network protocols that create network connectivity between systems. Although we discuss techniques for controlling movement in order to maintain connectivity within an ad hoc network, we do not discuss ad hoc network routing protocols.

1.5 Contribution

This thesis presents a distributed policy-based self-management framework for mobile autonomous systems. The framework enables autonomous systems of varying scale to perform self-management individually and as a team. The contribution of this thesis can be summarised as a novel policy-based approach to manage mobile autonomous systems such as mobile robots, an adaptable and reusable approach for mission specification, a novel management protocol to form and maintain a dynamic team, an optimal approach for role assignment, and communication link maintenance algorithms. The management framework uses three levels of mission specifications, namely, policy specification, mission class specification and mission class instantiation specification in order to enable reuse of policies and the mission classes. As a means of decentralising discovery and role management, the autonomous systems in a mission are arranged in the form of a management tree during the role assignment process. This tree is used for defining management hierarchies as well as data aggregation during execution of the mission.

1.6 Outline of the Thesis

The rest of the thesis is organised as follows. In Chapter 2, a background study of autonomic computing and policy-based systems management and related work on robot control, multi-robot and multi-agent architectures, with emphasis on mission specification, team formation and mission (task) allocation is presented.

Chapter 3 presents a mission specification approach for teams of mobile autonomous systems. We present a conceptual model of a mission and show how we use that model to specify missions in terms of roles and the relationship among them.

Chapter 4 presents a dynamic team formation approach in accordance with the mission specification. We describe how autonomous systems are discovered, admitted and optimally assigned to the mission roles. A novel management protocol used to form and maintain the team and a failure management scheme are presented.

Chapter 5 presents an approach for communication link maintenance among mobile autonomous systems. We present two complementary algorithms, which try to

control the movement of the autonomous systems in order to maintain the communication links and set up a rendezvous when the former fails or is infeasible.

Chapter 6 illustrates the applicability of our framework using a search and rescue scenario. We consider a mission to search and rescue survivors and assess damage in the aftermath of an earthquake disaster. We show (1) how the mission can be specified, (2) how the team is formed in accordance with the mission specification, and (3) how the mission is adapted. We also show how the specification can be reused for a similar but larger rescue mission.

Chapter 7 describes details of the implementation of the framework while Chapter 8 presents evaluation and a critical analysis of the framework.

Finally, Chapter 9 concludes the thesis with a summary of the achievements and directions for future work.

Chapter 2

Background

2.1 Introduction

In the previous chapter, we have defined an autonomous system as one that is capable of accomplishing its mission without human intervention by managing its resources and behaviours using its self-management (autonomic) capabilities. In order to enable a group of self-managing systems to undertake a joint mission, there are some key issues that should be addressed by the self-management architecture. These are set as design requirements in the previous chapter. Specifying the mission, describing the capabilities of the systems so as to facilitate the assignment, assigning each system to a role (task) of the mission and adaptation to current context are the most outstanding issues. In this chapter, we start with work that has shaped today's autonomic computing paradigm – robot-software architectures – and then we present an overview of related work that deal with one or more of the key issues we have identified in the previous chapter and assess them from the perspective of our requirements.

2.2 Robot-software Architecture

In this section, we present robot-software architectures, which are targeted for single robots. Although our interest lies in multi-robot systems, because these systems are

based on the architectures for single robots, understanding the single-robot architectures helps us understand the multi-robot systems. Single-robot architectures have served as the basis for building multi-robot systems that achieve a joint goal without explicit external management. A great number of software architectures have been proposed for single robots, most of these architectures can be categorised in one of the three classes of robot-software architectures. We present an overview of these classes of architectures and consider representative architectures from each class.

A robot can be defined as a machine that has the capabilities sensing, planning and acting organised in some way to produce intelligence. These three capabilities are the commonly accepted ones [Mur00], although some suggest a fourth capability referred to as learning. A robot-software architecture (control architecture) refers to the way the robotics primitives are organised, sensory data is processed, and distributed through the system [Mur00]. There are three types of robot-software architectures, namely deliberative, reactive and hybrid.

2.2.1 Deliberative Architectures

Architectures that perform thinking before performing an action are called deliberative. These architectures are characterised by the sense-think-act sequence, a world model and decomposition of the robot control problem into functional modules.

If we look at the sequence from a software point of view, it will become perception-planning-action. The world model, which is updated based on perception, is a model of the environment in which the robot exists. It contains sensed information and an initial (previously acquired) knowledge base. Action is produced by reasoning from the model. Figure 2.1 illustrates an architecture that uses the deliberative approach. The directed lines indicate the direction of information flow.

Deliberative architectures are also known as hierarchical because in almost all deliberative architectures the deliberation is performed in a stack of levels (from high to low) where higher levels create subgoals for the lower levels.

Using a global world model has some associated difficulties. Does a robot need to reason about the whole world just to move from point A to a nearby point B? If not, how does it know which part of the world model to use? This problem of representing

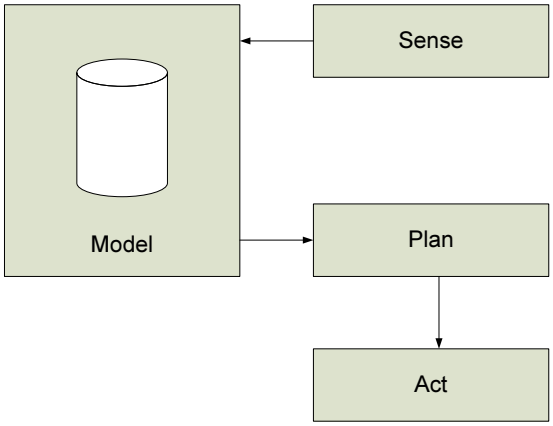


Figure 2.1: Deliberative Robot-software Architecture

the real world in a computationally tractable way is known as the frame problem. Although most of the hierarchical architectures try to solve this problem (or reduce its magnitude), by decomposing the planning part into layers where each layer uses a part of the world model, the decomposition is application specific and hence makes the solution not generic enough to be used in different domains.

The closed world assumption used in building global models, i.e., assuming that an object that does not exist in the world model to be non-existent in the real world incurs problems in a dynamic environment. Since the actions are a result of a reasoning from a closed world model, what the robot has to do when it faces an object that it was not supposed to face is unclear, and hence the robot will not function properly when (initially) a world model fails to capture all the details of the real world or when the real world suddenly changes. Table 2.1 summarises some selected deliberative architectures.

The main advantage of the deliberative approach is its ability to achieve a complex goal in less-dynamic and structured environments.

The main shortcomings of the deliberative approach are listed below.

- There is no direct coupling between perception and action. Due to this, robots of deliberative architectures are incapable of reflexive actions. The lack of reflexive actions causes a lack of real-time response (a cause for poor performance in uncertain environments).
- The robot has to always plan its next move, i.e., planning in every update cycle, which causes a computational overhead.

	Nested Hierarchical Controller
NHC [Mey93]	• Hierarchy of control loops; all performing the same operation at different resolution. • Decomposes the planning module into three levels (for example, mission planner, navigator, pilot); each level uses some portion of the world model. • Interleaves planning and action; if the world changes while the robot is executing a plan, it will stop and replan but the replanning is done only in the necessary planning level (for example, if the robot faces an obstacle on what was supposed to be obstacle-free path, it will replan its path, but the replanning will be done only at the navigator level).
	Real-time Control System
RCS [BAFH84, Alb93]	• A reference model architecture that defines the types of functions that are required in a real-time intelligent control system. • Partitions the control system into four elements, namely behaviour generation (task decomposition), world modelling, sensory processing and value judgement (cost, benefit and risk evaluation of a plan). • Layers of control systems (each containing the four elements) are stacked hierarchically (each level has a specific functionality). • In a similar manner to NHC, it interleaves planning and action. • Samples (extracts features) before integrating (fusing) the sensory information. • Has many versions with increasing complexity (RCS-1 up to RCS-4) [Alb93]. • Applied in various robot-application areas such as mining, cleaning, space and underwater exploration.

Table 2.1: Summary of Selected Deliberative Architectures

- Building a global model is not easy.
- If a higher level in the hierarchy fails, the robot will not "survive" as it does not know what to do. This is a result of the decomposition of functionalities in such a way that a lower level completely depends on a higher level (what a certain level has to do is always decided by its higher level, the level only decides how to do what it has to do).

2.2.2 Reactive Architectures

Architectures that are characterised by a direct coupling of perception and action (sense-act) and the absence of an intermediate planning unit are referred to as reactive. Since most architectures that use the reactive approach decompose the robot control problem into behaviours, they are also known as behaviour-based architectures. Architectures of this class vary in how they perceive behaviours and how they combine behaviours to produce complex behaviours. Table 2.2 summarises some selected reactive architectures.

The main advantages of the reactive approach are listed below.

- Fast response (since robots of these architectures are capable of reflexive actions).
- Ability to survive in dynamic and unstructured environments.
- Robustness (failure of one behaviour or behaviour-coordination does not necessarily render the robot dysfunctional).
- Convenience for incremental building and testing.

The main shortcomings of the reactive approach are listed below.

- Inability to achieve high-level (complex) goals as it lacks a planner. The sequence of behaviours a robot should execute to achieve a certain goal should be specified by the designer. The robot does not have the ability to select the appropriate behaviours and their execution sequence.
- Lack of performance monitoring. The robot has no means of evaluating its progress towards achieving the goal.

2.2.3 Hybrid Architectures

Hybrid architectures are architectures that use both deliberative and reactive approaches. Although the specific organisation detail differs from architecture to architecture, in general they are characterised by an upper deliberative layer and a lower

	Subsumption
[Bro86]	• Defines *level of competence* as a desired class of behaviours for a robot over all environments it will encounter. Some levels of competence: level 0: Avoid obstacle, level 1: Wander (without colliding), level 2: Explore (see places in the distance that seem reachable by heading towards them). • A single control system achieves one level of competence (e.g., one control system that enables the robot to avoid obstacles). It builds a control system for each level of competence where a lower level of competence is included in an upper level of competence (the lower level is a subset of the upper level). • When there is an input from the upper level to a unit in the control system of the lower level, the lower level prefers the input from the upper level than its own input from a preceding unit. In the absence of an input from an upper level, the lower level operates naturally (using its own output from the preceding unit as an input to the succeeding unit). • The Behaviour Language [Bro90] has later enhanced the capability of the subsumption architecture with mechanisms for behaviour grouping and communication between behaviours. Behaviours are modelled as a group of processes that have a common interface. The interface contains two types of elements, namely slots, which are accessible only by processes inside the behaviour and ports, which are accessible by other behaviours. Behaviours are composed by interconnecting ports with uni-directional connections.
	Motor-schema Based Control System
[Ark87]	• Behaviours are perceived as schemas that are composed of motor and perceptual schemas. Schema [LA89] is a way of expressing basic units of activity. It consists of both the knowledge of how to react and the way the reaction can be realised [Mur00, Ark98]. • Defines different schemas (e.g., stay-on-path, avoid-static-obstacle, find-land-mark, find-terrain; the first two are motor schemas and the rest perceptual schemas). It associates instances of perceptual schemas to instances of motor schemas to produce a behaviour (e.g., find-terrain associated with stay-on-path produces a staying-on-a-sidewalk behaviour). • Motor schemas are implemented as potential fields (e.g., avoid-obstacle is implemented with a repulsive potential field where the field is perceived as emanating from the obstacle). • Basic behaviours are combined by vector summation of the motor schema instances' potential fields to produce complex behaviours.

Table 2.2: Summary of Selected Reactive Architectures

reactive layer. By using both approaches, they achieve the planning capability of the deliberative approach and the reflexive capability of the reactive approach.

Figure 2.2 illustrates an architecture that uses a hybrid approach. The directed lines indicate the direction of information flow. The broken line between the planning and sensory unit is to indicate that the planning takes place once, and for a certain amount of time the sense-act sequence takes place without an input from the planning unit, although the planning unit might not have stopped planning.

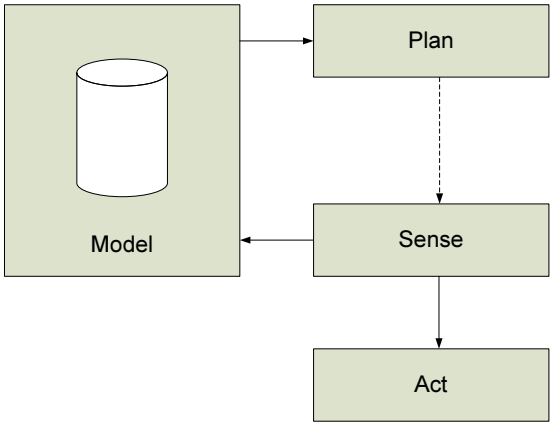


Figure 2.2: Hybrid Robot-software Architecture

Due to the fact that deliberative functions execute independently of the reactive ones, the planning is decoupled from real time execution. Sensory information is used by the reactive module to cause actions and by the deliberative module to build a world model. Some hybrid architectures use behaviours as the basic building blocks (bottom up approach) whereas some perceive the deliberative part as a starting point (top down approach).

The common components of hybrid architectures, as identified by Murphy [Mur00], are listed below.

1. Sequencer: generates set of behaviours in order to accomplish a task.
2. Resource manager: allocates resources to behaviours.
3. Cartographer: creates, stores and maintains spatial information including maps.
4. Mission planner: receives a command from a human and constructs a mission plan.
5. Performance and problem solving module: evaluates the robot's progress.

Murphy [Mur00] loosely classifies hybrid architectures, according to the way they organise the deliberative and reactive modules, as *managerial*, *state hierarchies* and

Name of Architecture	**Style**	**Remark**
AuRA [Ark87]	Managerial	Bottom up
SFX [MA92, Mur00]	Managerial	Bottom up
3T [BFG+97]	State hierarchies	Bottom up
BERRA [LOC00]	State hierarchies	Bottom up
Saphira [Kon97]	Model oriented	Top down
Wayfarer [Yam04]	Model oriented	Complete architecture not published.

Table 2.3: Summary of Selected Hybrid Architectures

model oriented. Managerial-style architectures are those which divide the deliberative module into layers based on scope of control or responsibility (e.g., mission planner, path planner, etc.).

State hierarchies use the knowledge of the robot's state to distinguish between reactive and deliberative activities where reactive activities are those which do not require knowledge of state (only present time) and deliberative activities are those which require knowledge of either past state (when executing a sequence of commands) or future state (when planning a mission or path).

Architectures with model oriented style use behaviours that have access to portions of a world model. Table 2.3 summarises some selected hybrid architectures.

2.2.4 Teleo-reactive Programs

A teleo-reactive (TR) program [Nil94] is an agent control program that directs the agent towards a goal, in a reactive manner. The program is a set of ordered condition-action rules as shown in Figure 2.3.

$$
\begin{array}{l}
C_0 \rightarrow a_0 \\
C_1 \rightarrow a_1 \\
C_2 \rightarrow a_2 \\
\vdots \\
C_n \rightarrow a_n
\end{array}
$$

Figure 2.3: TR Rules

The C_i are conditions evaluated over a sensory information and model of the world. The a_i are actions that are invoked if the corresponding conditions are satisfied. The

actions can be primitives or TR programs.

Conceptually, a TR program execution produces an electrical circuit and it is this circuit that is used for controlling the agent. This circuit semantics indicates that the conditions are evaluated continuously, and should there be a change in the environment, the change is reflected on the actions instantly.

The list of TR rules are scanned from the top and the first rule whose condition is satisfied gets its action executed. Should a higher level condition be satisfied while an action is being executed, the current action is stopped and the action corresponding to the higher level condition is started. This way the agent is directed towards its goal, i.e., the highest level condition.

The responsiveness of TR programs is similar to behaviour-based robot control approaches but TR programs have the added advantage of being responsive to stored models of the environment in addition to sensory inputs.

The major limitation of TR programs is that due to evaluation of all conditions continuously (or periodically, depending on the implementation), they involve much more computation than programs that check only relevant conditions.

Robot-software architectures, which enable robots to perform their tasks by their own, have self-managing capabilities although these capabilities are encoded in the control architecture and are not adaptable. As we will see in the next sections, these architectures, which are targeted for single robots, have served as the basis for building multi-robot systems that achieve a joint goal without explicit external management. Reactive (behaviour-based) architectures, in particular, have spurred a number of multi-robot systems because of their responsiveness and provision for elements, i.e., behaviours, which can be composed according to a higher level mission specification.

2.3 Mission Specification

Mission specification is the process of expressing a goal in a manner that autonomous systems such as robots and agents understand. In doing so, the goal is effectively decomposed into smaller goals, which can be achieved by computational elements of

the autonomous system referred to as tasks, behaviours and roles. The availability or absence of a mission specification mechanism can render an autonomous system easily programmable for diverse applications or bound to a specific application respectively. An autonomous system that does not allow for mission specification is usually designed for a specific application and it can only accommodate a single type of goal (application specific) with different parameters (e.g., assembly line robots used in manufacturing). Similarly, mission specification plays a crucial role in teams of autonomous systems by specifying the responsibilities of team members. It also serves as a means for explicit enforcement of team behaviours such as cooperation and response to team member failure. These behaviours depend on the specific mission and context; hence the ability to adapt them without changing the control architecture's code is necessary. The choice of the unit of specification such as behaviour, task and role also impacts the system's capability in allowing expressive (i.e., larger configuration space) mission specification and facilitating adaptation of the mission and reuse of the specification.

Cooperation in autonomous systems can be based on two models, namely emergent and intentional. In emergent cooperation, the systems have no intent to cooperate; the cooperation emerges while each system is performing its own task. In essence, the cooperation is choreographed by the designer of the control algorithms and subsequently most emergent cooperative behaviours appear only in homogeneous teams. In intentional cooperation, the systems are aware of the presence of other team members trying to achieve a shared goal and hence deliberately contribute to the activities performed to achieve the common goal. It is usually the case that the common goal (mission) is specified in a manner that allows the autonomous systems to take responsibility for parts of the mission but cooperate to facilitate each other's task.

In this literature review, we sometimes use the words coordination and cooperation interchangeably to preserve the original notion presented in some of the literature. However, we acknowledge the difference between coordination and cooperation as coordination does not necessarily involve sharing a common goal since a group of autonomous systems might be forced to coordinate, without any cooperation, by an external system such as a shared resource controller.

Research on mission specification for robots has resulted in different approaches that can be broadly divided into three types based on: (a) the application domain

they are targeted for – domain specific or generic, (b) the paradigm they use – plan based or specification (configuration) based, and (c) the number of autonomous systems involved – e.g., single or multi-robot missions. One can further divide these approaches, for example, a multi-robot mission could support homogeneous or heterogeneous robots. In this review, in line with our goal, we have considered multi-robot and multi-agent frameworks that address the issue of mission specification, are not domain specific and support heterogeneous systems.

Mackenzie et al. [MAC97] proposed an approach for mission specification that enables the organisation of a set of primitives in order to obtain a sophisticated system that can perform complex tasks. They have developed a language known as the Configuration Description Language (CDL) that can be used to specify the configuration of a robot or group of robots. The configuration is the specification of the components, connections and structure of the control system of the group. The low-level elements considered in this language, the primitive modules (behaviours), indicate what actions are performed but not how an actual action is performed, which makes the language robot-implementation independent.

The mechanism is developed for behaviour-based robots (i.e., those that use the reactive paradigm discussed in Section 2.2). The authors define an agent as *a distinct entity capable of exhibiting a behavioural response to stimulus*. Using this definition enables mapping each primitive capability of a robot to an agent. These agents are called atomic agents. Assemblage agents are recursively defined as *a coordinated society of agents* where the agents could be atomic or assemblage. Coordination determines how the society behaves, i.e., how it will react to a stimulus. The coordination can be competitive, temporal sequencing or cooperative. In competitive coordination, a subset of the society is selected to do the activation where the selection is based on some metrics. In temporal sequencing, a finite state machine that uses each agent's behaviour as a state is constructed. The behaviour of this machine is then the behaviour of the society. In cooperative coordination, each agent's behaviour is assigned a vector and weight and then the vector sum represents the society's behavioural consensus.

CDL is written and compiled using the MissionLab development environment, which also enables graphical design and simulation of a mission. Using CDL, a designer can define assemblage agents for different tasks in terms of the primitive behaviours.

```
/*Define cleanup behaviour as a prototype*/
defProto movement cleanup()
/*Instantiate three cleanup agents*/
instAgent Io from cleanup()
instAgent Ganyemede from cleanup()
instAgent Callisto from cleanup()
/*Create a janitor society*/
instAgent janitor from IndependentSociety(
      Agent[A] = Io,
      Agent[B] = Ganymede,
      Agent[C] = Callisto);
janitor;
```

Figure 2.4: An Example CDL Mission Specification [MAC97]

The assemblage agents are then mapped into a group of robots either manually or using MissionLab, which performs a static matching of assemblage agents to available robots in a repository. An example CDL specification for a cleaning mission is shown in Figure 2.4. This mission, referred to as *janitor*, is an assemblage of a primitive behaviour referred to as *cleanup*, which is undertaken by three robots. CDL allows for reuse of mission specification. The primitive behaviours specified in CDL are fully reusable and the assemblage behaviours can also be reused with little or no modification depending on the cardinality of robots as the specification commits to the number of robots during specification. As shown in Figure 2.4, CDL uses functional notation to specify the composition of behaviours, which makes it powerful with respect to expressiveness. CDL lacks a means of specifying adaptation at the mission level since all operational logics of the mission are included in the primitive behaviours and these behaviours cannot be changed during mission execution. One can, however, design the assemblage of behaviours with sufficient redundancy so as to enable graceful degradation when robots fail during the mission execution.

CHARON [AGH+00, ADE+00] is a language used for modular specification of interacting systems that has been used to describe multi-robot (agent) missions. CHARON models a system in terms of agents, and the system's behaviour in terms of hierarchical state machines referred to as modes. A single system is modelled as a composition of atomic agents whose behaviour is specified using modes. For example, a robot is modelled as a composition of the atomic agents, namely sensing, control and actuator. The behaviour of each of these atomic agents is specified using modes, which can also be composed of atomic modes. For example, the control atomic agent may be specified with modes such as *AvoidObstacle* and *FollowWall*, which could be

composed from atomic modes such as *MoveLeft* and *MoveRight*. A multi-robot system is modelled as a group of agents communicating through shared variables. A multi-robot mission is then specified using modes and transitions between modes of each agent representing the robot. Modes and agents are specified using a high-level language similar to a structured programming language and their composition is specified using set theory operations. Complex multi-robot missions are developed through sequential and hierarchical composition of modes and parallel composition of agents.

CHARON allows for reuse of both agent and mode specifications through instantiation with different parameters. Although CHARON is similar to CDL in that agents and their behaviours are specified using state machines, it significantly differs in its use of both discrete and continuous variables and constraints to control state transitions. The provision for continuous variables, for example, enables suitable specification of kinematical behaviours such as motion control. CHARON also provides a means for specifying access control by allowing variables in agents to be defined as *read*, *write* and *private*, and recovery from internal failure of an agent's mode through a mechanism called *history retention*, which enables rolling back to previous state. It, however, does not take into consideration adaptation of the mission when the context changes.

ALLIANCE [Par98] is a multi-robot architecture that supports the specification of adaptive missions using a behaviour-based approach where adaptation is based on mathematically modelled motivational behaviours, namely *impatience* and *acquiescence*. It groups behaviours that achieve lower level goals such as obstacle avoidance to form pre-specified higher level behaviours referred to as behaviour sets, which are capable of achieving higher level goals such as mapping. A robot activates a behaviour set at a time and commits itself to the goal associated with the behaviour set. This behaviour arbitration is performed by motivational behaviours. Each behaviour set has a motivational behaviour that controls its activation. Each motivational behaviour receives input from sensors, communications and other behaviours, and computes a motivation level that is compared against a threshold to decide whether to activate the behaviour set or not. When the behaviour set of a robot is activated, the robot has effectively allocated itself to the task corresponding to that behaviour set. A mission is modelled as a composition of higher level goals and hence can be

specified by stating which behaviour sets need to be active.

The *impatience* motivational behaviour is used by the robot to adapt to situations where other robots fail or under-perform and the acquiescence behaviour is used to adapt to situations where the robot itself fails or under-performs a given task. This approach gives greater autonomy to robots to decide on their actions and results in an adaptation scheme that transparently supports both fault tolerance and task reassignment for performance purposes. Motivational level thresholds for the pre-specified behaviour sets, which are central to the ALLIANCE mission specification approach, can themselves be adapted through a learning mechanism referred to as L-ALLIANCE [Par96]. These levels can be reused in similar missions to provide a better set of active behaviours. Compared to the behaviour-based specification approaches such as CDL and CHARON, ALLIANCE has a more elaborate implicit adaptation mechanism to both change in context and failure of robots. Explicit mission adaptation can also be specified by setting the motivational level thresholds. However, ALLIANCE's (initial) mission specification mechanism is less expressive (smaller configuration space) than these two systems since the mission is effectively specified by setting a single parameter (the motivational level threshold) to select one from a set of pre-specified behaviour sets.

AYLLU [Wer00] is a behaviour-based distributed robot control framework that is based on the subsumption architecture [Bro86] and the Behaviour Language [Bro90] (both discussed in Section 2.2). Behaviours are modelled as a group of processes that have a common interface with ports that are externally accessible. Tasks (missions) are formed through composition of behaviours by connecting ports from different behaviours using a uni-directional data path. Behaviours residing in different robots can be interconnected through their ports to form a distributed mission. Missions, behaviours and processes are specified, in a reusable manner, using a C-based language. AYYLU's processes are comparable to CDL and ALLIANCE's behaviours, and CHARON's modes. The behaviours are comparable to CDL and ALLIANCE's agents, and CHARON's behaviours. AYYLU's behaviour arbitration is similar to CDL, CHARON and ALLIANCE in that it inhibits or activates a behaviour out of a set of behaviours to provide functionalities required by the mission. There, however, is a fundamental difference between AYLLU's behaviour composition in that it allows hierarchical composition of behaviours residing in separate robots unlike the others,

which encapsulate behaviours (modes) in agents and allow only parallel composition of agents residing in separate robots. Mission adaptation to changes in context and failure are achieved through a dynamic task allocation approach referred to as BLE [WM00] (discussed later in Section 2.5).

The architectures we have seen so far, despite the presence of the ubiquitous element *agent* in their mission models, have been targeted for multi-robot systems. Mission specification has also been researched in the field of multi-agent systems. Tambe et al. [TPC00] proposed a framework for specification and monitoring of a robust agent organisation where the team behaviour is encoded in a wrapper entity referred to as TEAMCORE. TEAMCORE is targeted for agents with hybrid robot control architectures (Section 2.2). It employs reactive plans, which are pre-compiled plans with the ability to respond quickly in a similar manner to behaviours. TEAMCORE differentiates between team organisation and team goal. A mission designer specifies the team organisation as a hierarchy of goals where the leaf nodes are roles that are responsible for achieving all the goals that lead to their node in the hierarchy (their parent nodes). TEAMCORE's team behaviour specification is based on the STEAM [Tam97] teamwork model, which is built on the concept of joint intentions [LCN90, CL91]. A mission is specified using four sets of rules, namely joint intention, coherence constraints, intention tracking, and information-dependency relationship. The joint intention rules define the team's mental state upon which a joint commitment is defined to achieve a team goal until the team members believe the goal is achieved, or the goal is un-achievable or no longer relevant. The coherence constraints enforce team members to follow a common solution path so that they do not hinder each other's effort to achieve the joint intention. Intention tracking is a means of specifying responsibilities of team members for monitoring the activities of their peers. The information-dependency relationship rules enable an explicit declaration of the type of information that should be communicated among agents based on their dependency relationship. TEAMCORE is powerful in its ability to specify a mission in a flexible and reusable way as well as allowing adaptation easily through the addition of new intentions to the team. Its ability to specify communication constraints enables controlling the communication overhead introduced by teamwork albeit at the cost of the agents' freedom for autonomous behaviour. In addition, a TEAMCORE mission specification can be reasoned over to infer sub-missions, which

enables the mission administrator (human) to specify the mission at a much higher level in terms of intentions and generate sub-missions that can be undertaken by different sub-teams. TEAMCORE, however, does not exploit this capability to form sub-missions and allocate them to sub-teams as it does not support hierarchical organisation of roles. Although the framework has the concept of team hierarchy, this is a goal decomposition where the leaf nodes are roles and in effect all roles are managed centrally. TEAMCORE's employment of the joint intentions method makes it robust on the one hand, since as long as a goal is achievable, it will eventually be achieved even in the face of context change and failure. However, it is less reactive (slow) to context change and failure since all team members have to reason and agree before they abandon the present solution path and enter into a new one. This issue is especially crucial during failure in time sensitive applications and in applications where context change is frequent.

Tidhar's [Tid93] concept of social structures is similarly based on beliefs, goals and intentions collectively referred to as *attitudes*, and differentiates between team organisation and behaviour in its specification. A social structure (σ) is defined over a finite set of teams (τ), which itself is defined as a finite set of teams or individual agents. A team (τ^σ) is then defined as a team-set that believes (clarified later) a social structure (σ) as a team. A social structure is defined as a pair of teams ($< CD_\tau, CT_\tau >$) where CD_τ is the command team and CT_τ is the control team for the team-set τ. The command team is responsible for adopting joint goals, deciding on how to achieve them and achieving them. The control team is a subset of the command team and deals with coordination and control of the sub-teams so that they can execute the plan chosen by the command team to achieve the joint goals. An individual agent is defined as a special case of a team where the command and control teams of the agent are the agent itself. Attitudes held by agents are specified using first order logic and attitudes held by the team, i.e., joint beliefs, joint goals, and joint intentions, are specified using three modal operators, namely $MBEL$, $JGOAL$ and $JINTEND$. For example, $MBEL(\tau^\sigma, \phi)$, where ϕ is a first order logic formula, has the semantics: the mutual belief ϕ is believed by every sub-team in the team τ^σ and every sub-team believes that the mutual belief is held by the whole team. This approach shares the advantages of TEAMCORE as well as the limitations, i.e., the lack of immediate automated response to context changes and especially to failure. It, however,

provides a more general framework of mission specification in multi-agent systems where the team can range from central to completely distributed organisation. Hierarchical organisation can be achieved by enforcing the command team of every team to be a proper subset of the team and fully distributed organisation can be achieved by enforcing the command and control team of every team to be the team itself, which makes all agents have equal stand. The separation of the command and control team limits the scope of knowledge agents are required to have and also minimises communication by clustering the agents that need to communicate frequently. TEAMCORE addresses this issue by constraining communication using its information-dependency relationship rules.

Multi-agent system analysis and design frameworks have also dealt with specifying missions and organisations of agents to assist the design of multi-agent systems. Unlike the frameworks we have considered so far, these frameworks are targeted for guiding multi-agent systems design or assist in specification instead of themselves serving as a multi-agent systems framework. Nonetheless, their specification approach is of interest to us.

OMNI [DVSD04, VSDF05] is a multi-agent system specification framework composed of three dimensions, namely normative, organisational and ontological. It is based on two formal multi-agent specification frameworks [VSD03, DVSD04] that focus on the organisational and normative issues of multi-agent systems. The organisational dimension of OMNI deals with specifying the structure of the agent society while the normative dimension addresses the specification of norms that the agents in the society should abide by. The ontological dimension deals with contextual and communication aspects of the agent society. An organisation is modelled as a set of three elements, namely values, objectives and context. Objectives represent the overall goal of the society and they are specified using predicates. Values dictate what rules need to be defined for the organisation. The context, which can be left empty, specifies the set of organisations that exist in the environment where this organisation operates and have influence upon it. A mission in OMNI is then specified in terms of social structures (roles), scenes, social contracts and interaction contracts. Social structures are roles with objectives, rights and rules that are derived from the objectives and values of the organisation respectively. A role can be specified as either internal (institutional) or external, where an internal role is one that is responsible for coor-

dination and is enacted only by the agents representing the organisation while an external role is one that enters the organisation to help realise the goals. For example, in a conference organisation whose goal is to organise a successful conference, organiser and session chair roles could be internal roles while author, program committee member and presenter roles could be external. Figure 2.5 shows an example of a program committee member role in an OMNI mission of a conference society. OMNI specifications also have a set of obligations and abstract rules, described in deontic logic formulas, referred to as norms. Norms can be included in the specification of the different elements of the organisation. Rules, which are operational forms of the norms, are derived from the abstract norms and are described using dynamic logic. Scenes are processes performed by roles. Social contracts specify bindings of agents to roles and interaction contracts specify relationships between roles. OMNI's explicit representation of the team behaviour in terms of rules enables modifying the team behaviour without changing the agents. It also differentiates between the roles and agents enacting roles allowing norms to be specified without the knowledge of the agents enacting them.

Gaia [WJK00] models agent societies as organisations consisting of interacting roles and presents an approach for designing multi-agent systems in terms of roles. A role is modelled by using its responsibilities, permissions, activities and protocols. Permissions are the role's rights and activities are computations associated with the role that can be done without interaction with other roles. Protocols define the way the role interacts with other roles. Roles are enacted by agents and the agents enacting the roles are tied to the roles at design time. Gaia also considers static norms with respect to agents and hence roles. In [DWS01], a similar approach to Gaia, with a more elaborated approach for identifying the roles necessary for an organisation, is presented. However, both approaches do not provide a means to explicitly specify organisational structures and lack organisational norms that enforce global behaviours. SODA [Omi01], a multi-agent systems design methodology based on Gaia, tries to partially address this shortcoming by using tasks as building blocks of agent societies where the tasks are divided into individual and social ones. Individual and social tasks are assigned to an individual role and a group respectively where a group is made up of social roles.

In this section, we have discussed different multi-robot and multi-agent architectures

Id
PC_Member
Objectives
*/*The desired result of the role*/*
$paper\_reviewed(Paper, Report)$
Sub-objectives
*/*Landmarks for the objective*/*
$read(P), reported(P, Rep), review\_received(Org, P, Rep)$
Rights
*/*This role has the right to access the conference manager program*/*
$access\text{-}confmanager\text{-}program(me)$
*/*PC_Member is OBLIGED to understand English*/*
*/*IF paper_assigned THEN PC_Member is OBLIGED to review the paper BEFORE the given DEADLINE*/*
*/*IF author of paper_assigned is colleague THEN PC_Member is OBLIGED to refuse to review as soon as possible*/*
Norms
$O_{PC\_member}(understand(English))$
$done(assign\_paper(P, me, Deadline)) \rightarrow$
$O_{PC\_member}(review\_paper(P, Rep)) < do(pass(Deadline)))$
$done(assign\_paper(P, me, \_)) \wedge is\_a\_direct\_colleague(author(P)) \rightarrow$
$O_{PC\_member}(review\_refused(P) < pass(TOMORROW))$
Rules
$done(assign\_paper(P, me, Deadline)) \wedge \neg done(review\_paper(P, Rep))$
$\rightarrow [pass(Deadline)]V4$
$done(assign\_paper(P, me, \_)) \wedge is\_direct\_colleague(author(P)) \wedge$
$\neg done(review\_refused(P)) \rightarrow [pass(TOMORROW)]V5$
Type
$external$

Figure 2.5: A Role in OMNI's Mission Specification [VSDF05]

that enable specification of missions. These frameworks are summarised in Table 2.4.

In general, the frameworks that are targeted for robots tend to specify the mission in a bottom-up fashion by starting from the primitive behaviours and composing these to provide complex missions. The strength of these frameworks lies in the reflexive nature of the primitive behaviours, which makes the complex missions responsive enough so as to be used in environments involving frequent changes. The absence of separate rules of adaptation and explicit specification of team structures is noticeable in these architectures. This makes them less suitable for our purpose, i.e., mobile autonomous systems targeted to operate as a team, with clearly defined structure, and adapt to context changes. The frameworks targeted for agents tend to follow a top-bottom approach by starting from the overall goal of the team and specifying the mission as a decomposition of this goal. A rather elaborate organisation of agents into teams and specification of separate rules for team and agent behaviours is noticeable

	Specification	**A**	**R**	**T**	**S**
CDL [MAC97]	• Composition of behaviours • Functional notation	N	Y	N	N
CHARON [AGH+00]	• Composition of modes (hierarchical state machines) • Similar to a structured programming language	N	Y	N	Y
ALLIANCE [Par98]	•Setting motivation level threshold for pre-specified behaviour sets	Y	Y	N	N
AYYLU [Wer00]	• Composition of behaviours • High-level (C-based) language	Y	Y	N	N
TEAMCORE [TPC00]	•Sets of rules: joint intention, coherence constraints, intention tracking, and information-dependency relationship • First order logic based • Has the concept of roles	Y	Y	N	N
Tidhar [Tid93]	• Sets of rules: beliefs, goals and intentions • First order logic based • Has the concept of roles	Y	Y	Y	N
OMNI [VSDF05]	•Social structures (roles), scenes, social contracts and interaction contracts •Set of obligations and abstract rules described in deontic logic •Concrete rules described in dynamic logic	Y	Y	Y	Y
GAIA [WJK00]	•Static agent societies in terms of roles •Propositional logic based norms for roles	N	Y	N	Y

A: Adaptation **R**: Reuse **T**: Team-structure specification **S**: Security (authorisation)
Y: Yes **N**: No

Table 2.4: Summary of Multi-robot and Multi-agent Frameworks that Support Mission Specification

in these architectures. The explicit specification of team behaviours in terms of rules is a useful approach for our system. However, as these rules are specified in computationally hard models we need to look for alternative forms. Security specification, in the form of authorisation, is also considered in some of the architectures.

Once a mission is specified, autonomous systems need to be assigned to the roles or tasks (or any unit of specification of the associated framework) in order to execute the

mission. The decision of which autonomous system to assign to which role cannot be made in an efficient way without knowing the capabilities of each autonomous system and matching them with the capability requirements of the role or task. In the next section, we review related work on capability description and matching.

2.4 Capability Description and Matching

A capability description defines the actions an autonomous system can perform, e.g., move from A to B or pick up an object, or services/resources it can provide related to its sensors such as infra red imaging, chemical detection, video streaming, etc. This description is necessary when role (task) allocation is performed in a joint mission involving multiple autonomous systems in order to decide what role (task) a system should be assigned to among a set of mission roles (tasks). There has been research in capability description and matching for service composition, and adaptation as well as task allocation in multi-robot and multi-agent systems. In this section, we consider related work that deal with capability description and matching.

2.4.1 Capability Description

In [And02], a simple capability description, which is intended for use in the Session Description Protocol (SDP), was proposed. SDP is used to initiate multimedia sessions. In this description, the service does not allow the user to perform operations. The capability description is used only to adapt to the service or to negotiate parameters of the service. A user can indicate preferences in terms of specified parameter values such as audio or video format so as to get the service provided with a characteristic of the preferences. As a result, this service description can efficiently be done as a listing of attributes of the service and their possible values. Of course, there is a need to agree on the attribute names and semantics, i.e., defining an ontology. This type of description is sufficient for describing the capabilities of autonomous systems provided that once each system has been assigned to its role (task) there is no need to perform remote operations. Although it is applicable, for some scenarios that do not involve remote interactions, this will be too restrictive for our system.

Some service description mechanisms integrate request/reply messages with the operations of the service. In this case, the operations are building blocks for the request/reply protocol instead of an operation in a service interface. SPML (Service Provisioning Markup Language) and WSDL (Web Services Description Language) are two such service description mechanisms.

SPML [OAS03] is a service-description protocol developed by the OASIS Provisioning Services Technical Committee (PSTC). It defines an XML-based framework for exchanging user, resource, and service provisioning information. It is a request/response protocol where the requests and responses are well-formed SPML documents. A resource (Provisioning Service Target) describes its services using an SPML compliant service interface. A client (Requesting Authority) sends requests to the server (Provisioning Service Point) in the form of SPML. The server communicates with the resource and sends the results back to the client. In some cases, the server can be the resource itself.

In WSDL [CGM+04], services are conceived as a collection of end points (ports). A port is an instance of a port type – an abstract set of operations supported by one or more endpoints. When a port type assumes a specific protocol, referred to as concrete protocol, and a data format specification, it becomes a port. This process is termed as binding. An operation is an abstract description of an action supported by the service. WSDL also has the concept of messages as an abstract, typed definition of the data being communicated. Types are containers of a data type definition. A service in WSDL is then described by an XML based definition of six main elements, namely types, message, portType, binding, port and service.

SPML and WSDL can serve the purpose of describing an autonomous system's capability including the operations thereby allowing remote invocations across systems. However, because these protocols are designed for large scale composition of services the overhead of employing these systems, especially when capability descriptions are communicated, would outweigh the benefit for our system. This is because our system would have most of the autonomous system's tasks contained locally, with occasional remote operations to use services that are available in other autonomous systems.

The Universal Plug and Play (UPnP) Forum has been developing capability description

schemes for devices and services [Upna, Upnb]. A UPnP device description is an XML document that has three main parts. The first part is a description of the device's parameters such as its type, model, manufacturer, etc. The second part is a list of services provided by the device, which contains pointers to corresponding service descriptions. The third section is a list of devices embedded in the device.

A UPnP service description contains a list-of-actions and a service state table. The list-of-actions section contains a listing of the operations that can be invoked by a service user. The operations description specifies the name of the operation, the arguments (if any) and the related state variable (if any) to each argument. Specifying the related state variables is an indirect specification of allowed values to arguments. In addition, some operations have a direction element showing whether the operation is reading or setting a state of the service. The service state table is a list of state variables, their data types and allowed values. If a service generates an event when its state variable changes, the state variable will have an attribute that shows that it sends events.

A UPnP type description would be able to describe the different components of autonomous systems such as robots (e.g., sensors) and could be adapted to describe services and associated operations with autonomous systems. Its concise approach makes it appealing for our system, where capabilities need to be communicated instead of being stored in a registry.

2.4.2 Capability Matching

LARKS [SKWL99, SWKL02] is a capability specification and matching language for agents where an agent can be a service requester, provider or a middle agent such as a matchmaker. Provider agents describe their capabilities using LARKS and advertise them to middle agents, which store the advertisements. Requester agents ask middle agents for a provider that has the desired capabilities. The middle agents perform the matching and reply with a subset of the stored advertisements that satisfy the requirement to the requester agents.

A LARKS capability specification contains the context of the specification, input and output variable declarations and logical constraints on these variables. It also may

include the optional definition of data types, description of the meaning of words used in the specification and textual description of the specification. In LARKS, domain ontology is written using ITL (Information Terminological Language) [SLK98] and concepts are defined as a conjunction of logical constraints that should be satisfied for any object to be an instance of that concept. Concepts can be attached to the context, data types and the description of meanings. Requests and advertisements have the same format and are differentiated by the information included in the Knowledge Query and Manipulation Language (KQML) [FFMM94] message in which they are wrapped up.

LARKS considers three types of matching, namely *exact*, *plug-in* and *relaxed* match. An exact match is where the request and advertisement are either literally equal or one can be obtained from the other using a renaming of variables or logical inference. A plug-in match is a less accurate match in that the advertisement can be used to satisfy the request, yet the two can differ in their input and output declarations as well as constraints. A relaxed match is the least accurate match since it does not assure that the request and advertisement match semantically; it determines the closeness between the two in terms of a numerical distance value. The matching process is done using a series of increasingly stringent filters, namely context matching, profile comparison, similarity, signature and constraint matching. The filters are independent from each other and the quality of the resulting match varies depending on which filters are considered.

In [PKPS02], a service description language based on DARPA [1] Agent Markup Language (DAML) [HM00] and DAML-S (DAML for services) [ABH+02], and a semantic-based matching between advertisements and request for web services is proposed. Advertisements and requests are described as DAML-S profiles and the semantic matching is based on DAML ontology in that the advertisements and requests refer to DAML concepts and the associated semantics. An advertisement is said to match a request when all the inputs of the advertisement are matched by the inputs of the request and all outputs of the request are matched by outputs of the advertisement. The matching algorithm differentiates between four degrees of matching, namely *exact*, *plug-in*, *subsumes* and *fail*. For example, when matching the outputs of a request (O_R) and advertisement (O_A), if $O_R = O_A$ or O_R *is subclass of* O_A then the matching

[1] Defence Advanced Research Projects Agency

is *exact*, if O_A *subsumes* O_R the matching is *plug-in* and if O_R *subsumes* O_A then the matching is *subsumes*, where the advertiser does not completely fulfil the request.

Capability description and matching facilitate the assignment of autonomous systems to roles or tasks by providing a measure of suitability of a system for a given role. In a situation where the autonomous systems that are going to form the team are known beforehand and the team is static, a mission administrator can (manually) check the capabilities of the systems against the roles, assign the capable systems to the roles and start the mission. However, in a team of mobile autonomous systems where members are not known in advance and the team is dynamic in that members join and leave during the mission execution, the ability to describe and match capabilities does not suffice. The ability to discover autonomous systems, match their capability with the mission roles (tasks) and assign them accordingly is necessary. In order to cope with changes in the mission, whether due to changes in strategy, change in context, or loss of existing roles, the ability to reassign roles is also important. In addition, in both static and dynamic teams the assignment should be optimised to best utilise available autonomous systems. In the following section, we present related work in role or task allocation in multi-robot and multi-agent systems. Because the problem of optimal task allocation has been studied in the area of systems deployment, we will also consider relevant work from that area.

2.5 Role Assignment

Allocating roles (tasks) to multiple robots in an optimal way is a crucial element of cooperative autonomous system architectures. This problem, in the most general case, is NP-hard [Par94]. Also, in applications targeted for time sensitive missions, in addition to the optimality of the assignment, the time taken to compute the assignment should be taken into consideration. In addition, the approach should be able to scale with the size of the team. Several approaches have exploited domain specific properties and employed approximative methods to solve this problem. The role (task) allocation problem in multi-robot and multi-agent systems is related to the more general problem of optimal deployment and reconfiguration of systems in a networked environment. A number of role/task/system allocation (deployment) schemes exist for multi-robot, multi-agent and distributed systems deployment ap-

plications. These approaches can be broadly classified as those which use utility functions, and those which use planning or constraint satisfaction approaches to search among the space of possible assignments. In this review, we have presented representative architectures from both classes of approaches with a bias towards approaches that are targeted for multi-robot and multi-agent systems and deal with dynamic role/task allocation either initially or during reconfiguration.

In [AHW03, AHW07], a framework for optimal deployment and reconfiguration of systems on a dynamic networked environment is presented. The deployment and reconfiguration process is perceived in a similar manner to the *sense-plan-act* process in robot-software architectures (Section 2.2). The framework has two elements – a configuration manager that deploys and reconfigures the systems and a planner that generates the configuration. The configuration manager, called Planit, deals with the sensing and acting parts of the process in that it monitors the system and obtains events from which it generates the current state of the system. If there is a need for a reconfiguration, the current state with a desired state that is generated according to the configuration rules is used to replan. Planit provides the domain and configuration specifications, which are specified using the Planning Domain Definition Language (PDDL), to the planner. The domain contains the types of entities involved in the deployment (components, connectors and machines), predicates (constraints) associated with the entities (e.g., a predicate that indicates a certain component is placed on a certain machine), utility functions, and actions that can be included in the plan to change the state of the system. The configuration specification consists of the current state and goal state, which is the desired state of the system. The planner, using the LPG (Local Search for Planning Graphs) heuristics, generates sequences of actions that should be achieved in order to lead the system to the desired state and replies to Planit, which performs the actions on the system. Reconfiguration can be used to improve the quality of deployment as well as failure recovery. Planit can be used to perform task assignment for a wide range of applications since it allows domain definition. The framework has a dynamic nature due to its support for reconfiguration during changes in goal or network connectivity. It, however, assumes that there are a fixed initial set of components (to be deployed) and does not allow the addition of new components. Also, the planning process is computationally intensive resulting in a longer assignment time. For example, to decide the placement

of 60 components on 10 machines, the planner (running on Sun Ultra/2200/512 with two 200MHz CPUs, 512MB RAM) was able to find a plan in 412 seconds due to the large search space. The authors report that the maximum number of components they have experimented with is 120. This approach is less suitable to role assignments for mobile autonomous systems because of its assumption of fixed sets of components and time consumption.

In [IHAK02, KIK03], a framework that consists of a declarative service specification in terms of components, support for component deployment on a network, and a planning module is presented. It enables services to be built up from distributed components and facilitates migration and replication of components transparently. The service specification and the current state of the network is provided to the planner, referred to as Sekitei [KIK03], which determines optimal locations for component placement. The planning module employs multiple planning techniques in that it uses regression and progression planning. The framework assumes that the network on which the distributed system is deployed is static with respect to the set of nodes and links as well as their properties. In addition, similar to Planit the planning process is computationally intensive resulting in a slower decision of placement.

COCoA [KNS06] formulates the problem of task allocation and scheduling for a tightly coupled (i.e., no robot can achieve a single goal by itself) team of robots in a search and rescue domain, using goals and constraints. Three constraints, namely goal, robot and resource constraints are specified using first order logic. The constraint optimisation problem is modelled as a Mixed Integer Linear Programming (MILP) problem and CPLEX [Inc] is used to solve it. Since finding a feasible solution might take a considerable amount of time (hours even days for large search and rescue problems), different heuristics are used to improve the solution time.

In [MMRM05, MRMM05], an approximative and decentralised approach for determining a distributed software-system's redeployment that maximises its availability is presented. The system is modelled as a set of components and their properties, a set of hosts and their properties, a set of constraints, the system's initial deployment as a mapping of components to hosts and a set of system properties that are visible from a given host. A utility function that describes the system's availability as the ratio of the number of successfully completed interactions in the system to the total number of attempted interactions is defined. The redeployment problem is then

defined as finding a function that maps the components to hosts so that the utility function is maximised without violating the constraints. The optimisation criterion considers frequency of component interactions and reliability of communication links between hosts. In contrast to Planit, Sekitei and COCoA, this approach considers only redeployment since it assumes the existence of an initial deployment. However, compared to these approaches, with respect to time taken to find a feasible solution, it provides a more scalable means for determining a distributed software-system's redeployment by employing a utility function based approach.

The Contract Net protocol [Smi80] provides a means for distributing tasks through negotiation. Each node in the net takes one of the two roles, namely manager or contractor. Managers announce tasks, and potential contractors submit bids to the managers. The managers then evaluate the bids and award contracts to the bidders. The contents of the negotiation messages are problem-domain dependent and the user is responsible for specifying the content. As we will see in the reviews ahead, the Contract Net assignment model has been applied in a number of multi-robot and multi-agent systems [UEWA07, WJK00, GM01, GM02].

MURDOCH [GM01, GM02] is a framework supporting inter-robot communication, through a publish-subscribe system, and dynamic task allocation to facilitate cooperation. Its completely distributed task allocation is based on auctions and contracts [Smi80, DS83]. A task is announced by an agent (one of the robots, a human, etc.) with the required resources being used as the destination address. Robots with these resources compute their fitness and announce the result, and the robot with the highest fitness takes the task. MURDOCH deals with task reassignment through progress monitoring and contract renewal, where a contract is renewed only if there is a sufficient progress by the winner of the auction. As a result of a completely distributed task assignment scheme, MURDOCH suffers the same problem as the greedy algorithms problem – equivalent to an instantaneous greedy scheduler, where decisions are made based on only the current and/or local situation without taking into account how the decision might affect the future and/or the global situation. In effect, these types of algorithms may not always give the best solution.

Iocchi et al. [INPS03] present an approach for coordination of robots based on dynamic role assignment. Their system is layered with a coordination protocol running on top of a communication protocol. The basis of the communication protocol is the

publish-subscribe paradigm with some support for low-level communication using a custom UDP. The coordination protocol is based on utility functions that are defined for each role. The mission administrator (manually) orders roles according to their importance (priority) and defines a percentage that denotes how many of the robots in the team should be assigned to each role. Each robot computes the utility for each role and exchanges the computed values periodically, and the robot with the highest value for a given role takes that role in accordance with the priority (i.e., high priority roles taken first). Adaptation is achieved through exchanging roles based on periodic broadcast of utilities performed by each robot. Although exchanging roles based on periodic utility broadcast and evaluation keeps the assignment optimal throughout the mission lifetime (at the cost of computation and communication overheads), it has the unnecessary effect of destabilising the team. In addition, because the robots self-assign themselves to a role, in the event of communication failure some or all capable robots may take the highest priority role since they will not be aware of other robots' decisions, resulting in a redundant and incomplete team configuration.

Similarly, in [CKC04], a hybrid-automata [Hen96] based paradigm for cooperating robots is presented. In this approach, hybrid automata are used to represent roles, role assignments and discrete variables related to each robot. Cooperation is modelled as a composition of these automata and role assignment is based on utility functions that compute the utility of a robot performing a given role. Adaptation is achieved through role reassignments and exchanges based on utilities. Since possible role reassignments and exchanges are defined a priori using a hybrid automaton, either the possible types of roles a robot can assume are limited or the hybrid automaton has to consider every possible types of roles a robot can play. This fact makes the approach applicable only to teams with limited number of robots and role types mainly for tightly coupled cooperation such as cooperative manipulation of objects.

In [SS], a Distributed Constraint Satisfaction Problem (DCSP) based solution for reassigning tasks to robots is presented. In this approach, it is assumed that each capability of a robot corresponds to a task the robot can perform, and the robots are initially given a role graph that is a Distributed Organisational Task Network (DOTN), where a role is perceived as a set of tasks. The robots then try to minimise remote task dependencies by searching for minimal dependency among tasks, using local

search approaches, and trading tasks and responsibilities (switching positions of a parent and a child in the task network). In this model, in addition to task dependency, there is an implicit utility that measures the fitness of the robot against the associated capability before the task trading is made. In a scenario where the mission specification does not group tasks that are likely to have more interaction in one element such as roles, this approach provides a useful solution. However, if the tasks are initially grouped, the communication overhead from the exchange of utilities between all robots and the computation overhead from the search may outweigh the benefit

Since we are interested in dynamic teams, the approaches we have considered so far are only the ones that deal with dynamic role or task allocation. We now reconsider some of the multi-robot and multi-agent architectures we have discussed in Section 2.3 and discuss their task allocation schemes.

In CDL [MAC97], either the mission administrator (manually) or the MissionLab tool performs the matching and assignment of behaviours to robots. This static matching and assignment of behaviours to robots makes the approach inapplicable to domains where the knowledge about the capabilities of robots may not be available until the mission execution time. Ulam et al. [UEWA07] built on CDL and proposed a mission specification system with a case-based reasoning approach for generating mission plans and a Contract Net Protocol [Smi80] based task allocation. In CHARON [AGH+00], the difference in capabilities of robots is not taken into consideration; hence there is no matching process and the allocation is done by the mission administrator (manually) similar to CDL. In ALLIANCE [Par98], the initial assignment is done by the mission administrator. Each robot then broadcasts its activity related to a task. Upon receiving this information, the robots compute their motivational behaviour and decide whether to take away the task from the robot currently assigned to the task based on the resulting motivational level. ALLIANCE's learning mechanism L-ALLIANCE [Par96] tracks which motivational level resulted in better performance and updates the motivational level threshold which in turn adapts the assignment process. AYYLU [Wer00] performs task allocation using an approach called BLE (Broadcast for Local Eligibility) [WM00]. BLE is based on behaviour arbitration where each BLE-arbitrated behaviour, in addition to the task related ports of AYYLU, has three ports, namely *Local*, *Best* and *Inhibit*. The *Local* port of each

behaviour broadcasts its locally-computed eligibility estimate to the *Best* port of the others, where eligibility is a measure of the fitness of a robot to a given task. The behaviour with the highest eligibility inhibits the other behaviours thereby effectively enabling the robot to allocate itself to the appropriate task. Since eligibility is computed periodically, the tasks are reassigned when there is a change in the eligibility ranking. Both ALLIANCE's and AYYLU's approach are similar to Iocchi et al.'s [INPS03] self-assignment discussed previously. TEAMCORE [TPC00] uses a special agent referred to as KARMA to discover agents, and the mission administrator (manually) performs the assignment of the discovered agents to roles. KARMA queries different agent naming service (ANS) agents, where agents are supposed to register, to discover agents and also monitors the team's progress during the mission. In Tidhar's [Tid93] social structure, although there is no assignment scheme employed, the importance of dynamic role allocation scheme is acknowledged, and it is indicated that schemes such as Contract Net could be used. OMNI [VSDF05] does not deal with role allocation, and since GAIA [WJK00] assigns agents to roles at design time, the allocation is static.

In this section, we have considered various approaches for dynamic role or task allocation and also reconsidered the architectures we have discussed in relation to mission specification in order to review their role allocation approach. Although the approaches are too diverse to summarise in one paragraph, we will try to factor out the most relevant issues to our objective. The systems deployment and/or role or task allocation approaches that use a planning or constraint satisfaction approach [AHW07, KIK03, KNS06] tend to scale less with the number of roles or systems involved, due to the large search space involved, and hence are not suitable for our purpose. Utility functions are used in most of the approaches [MMRM05, GM02, INPS03, CKC04, Par98, WM00] to measure the fitness of a system for a task/role either for initial assignment or reassignment. This approach is powerful provided that the utility function is chosen appropriately and hence is worth considering for our framework. In some of the architectures that support mission specification, either the matching or both matching and assignment are done by the mission administrator (i.e., manually). In some, a self-matching and self-assignment approach assisted with broadcast of utilities is used. This does not guarantee assignment of crucial roles to best available systems since there is no means of enforcement that dictates

the assignment other than the decision made by the robot/agent itself. It is also worth noting that the ability to discover new systems as they arrive is noticeably absent in these architectures. Hence, while we carry forward useful lessons and concepts to our design, none of these architectures is suitable to our purpose in its entirety.

Once a team is formed in accordance with a mission specification by assigning roles to autonomous systems that have the necessary capability, the mission should be able to adapt itself in order to cope with context change. In Section 2.3, when we discussed mission specification, we have observed that architectures that use explicit rules to specify the team behaviour allow adaptation without changing the code for the agents or robots. We have also observed that those that employ this approach use computationally hard models such as deontic logic to model these rules. We therefore consider the concept of using explicit rules and look into computationally less demanding forms of rules. Rule based systems management, i.e., policy-based management [Slo94], has long been used in distributed systems and proved to be an efficient approach. In the next section, we consider policy-based systems whose main goal is adaptive system management and hence relevant to our objective.

2.6 Mission Adaptation

Autonomous systems operating in an uncontrolled environment experience frequent changes in context. In order to cope with these changes, the systems should either have behaviours for all possible scenarios built in to their code or an external adaptation mechanism should be employed. In this section, we consider one such external adaptation approach, i.e., policy-based adaptation.

2.6.1 Policy-based Systems Management

A policy is a rule that defines how a system should behave under certain circumstances. Policy-based management is an approach where the management is based on rules (policies). In the context of policy-based management, a policy can be defined as a rule that controls access to a resource or a rule that controls actions

performed by a system. These two purposes are different and hence give rise to two fundamental types of policies.

Policy-based management enables automation of systems management and changing the behaviour of a system dynamically [Slo94], reduces complexity in management applications, and leads towards realising self-management [Ver02, DKFS02, WHW+04]. The power of a policy-based approach lies in its ability to separately define the choices in the behaviour of a system from the system's implementation, in a manner that is easy to define, comprehend and interpret. The approach, however, is not without challenges. When applied to larger systems that need more rules, tracking which policy does what becomes difficult and conflicting policies can arise.

There has been research on using policies in distributed systems to manage interactions among systems [MU00], configuration management [Bur95], access control [LMSY96, BMY02], trust management [LMW02], and overall behaviour of systems [Slo94]. In this section, we consider representative work on policy-based approaches that deal with both specification and enforcement of policies with a broader scope of management (in contrast to security, trust, quality of service, etc. management) and hence are applicable to wider domains of application. We discuss the main elements of these frameworks, i.e., the policy specification (representation) and deployment model (enforcement architecture) and assess their applicability to our purpose with respect to support for specification of adaptation as well as security (authorisation).

The IETF Policy Framework

The Common Information Model The Common Information Model (CIM) [DMT03] unifies and models all aspects of a managed environment using an object-oriented approach. It does so by defining common models for all entities involved in the managed environment such as networks, devices, systems, users, applications, etc., and the relationships between these entities. Each model is represented by an object-oriented format known as Managed Object Format (MOF).

The two main advantages of CIM are its object orientation and unification of models. It has the qualities of an object oriented system (abstraction, inheritance, etc.) and by consistently defining models of entities and the relationships between them it enables management applications to perform actions that can involve a number of entities

(for example, a diagnosis application that traverses all involved entities by using CIM relationships, and explores all entities by checking CIM objects representing the entities).

Policy Core Information Model The Policy Core Information Model (PCIM) [M+01], developed by the IETF (Internet Engineering Task Force) policy working group and the Distributed Management Task Force (DMTF) CIM activity, is an object oriented model for representing policy. This model has two hierarchies of classes, namely policy classes and association classes. Some of the classes in the policy hierarchy are PolicyRule, PolicyCondition, PolicyAction and PolicyGroup whereas some of the classes in the association hierarchy are PolicyConditionInPolicyRule, PolicyActionInPolicyRule, PolicyTimePeriodCondition, PolicyRuleInPolicyGroup and PolicyGroupInPolicyGroup.

In the IETF policy framework, a policy has an *"if condition then action"* semantics. The PolicyRule class represents this semantics associated with a policy. Strictly speaking, this class represents neither the actual condition nor the action; it only specifies how the condition and the action should be interpreted. The condition and action associated with a policy are represented by the PolicyCondition and PolicyAction classes respectively. The association of condition and action objects to policy-rule objects is represented by the association classes PolicyConditionInPolicyRule and PolicyActionInPolicyRule respectively.

Deployment Model The IETF policy framework deployment model has four main components, namely the policy-management tool, the policy repository, the Policy Decision Point (PDP) and the Policy Enforcement Point (PEP). Figure 2.6 illustrates these components and the interaction between them.

The policy-management tool is used to define and add policies to the policy repository where policies are stored. The IETF policy framework requires that all the policies that are stored in the repository should be in one of the information models specified in the PCIM so as to establish inter-operability.

The policy decision point is responsible for interpreting policies. It is defined as *a logical entity that makes policy decisions for itself or for other network elements that*

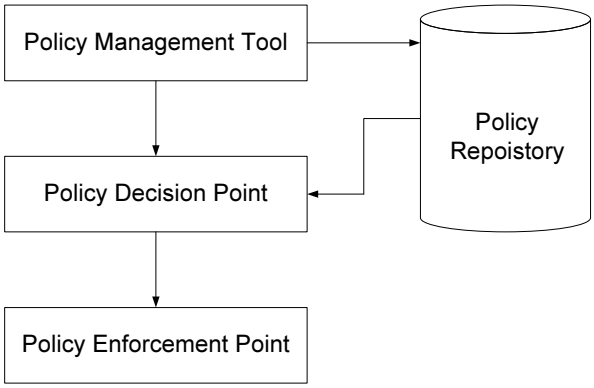


Figure 2.6: The IETF Policy Deployment Model

request such decisions [W+01]. The decision may be either evaluating conditions or dealing with actions that might need to be enforced based on the result of the condition evaluation. The policy enforcement point is the entity that actually enforces the actions. In the most general case, the PDP and PEP may be on different devices.

The IETF policy framework provides an information model to specify basic policies and groups of policies and leaves the definition of the concrete policy specification language to application domains. The modelling approach has the advantage of being easily mappable to structured languages such as XML, which makes it suitable for analysis and distribution of policies. The main drawback of this approach in relation to our objective is the lack of support for explicit authorisation policies.

The Ponder Policy Framework

Policy Specification Language Ponder [DDLS01, Dam02], is an object-oriented policy-specification language developed at Imperial College London. It provides a common language that enables administrators to specify a policy, hence unifying the concepts and models of various policy related research.

Ponder supports a family of policies named *Access Control Policies*, which are mainly used for security management, and *Obligation Policies*, which are mainly used for service management. Both policy types can be used in either type of applications. For example, obligation policies can be used for logging suspicious actions, which is a security-management issue.

Generally, in the Ponder policy framework, a policy has a *"when event occurs, if*

Type	Purpose	Enf.	Sign	Remark
Authorisation	Protect resources and services from unauthorised use.	Target	Both	
Information Filtering	Transform the information input or output parameters of an action.	Target	N/A	It can only be applied to positive authorisation actions.
Delegation	Temporarily transfer access rights.	Target	Both	It specifies the delegate and does not control the actual delegation/revocation of access.
Refrain	Restrain subjects from performing actions that they must not perform.	Subject	N/A	It has the same effect as applying a negative authorisation policy on a target. However, this is used when one does not trust the target.
Obligation	Specify the actions that must be performed by managers when an event occurs.	Subject	N/A	Event triggered.

Table 2.5: Summary of Basic Ponder Policy Types

condition then action" semantics where the event part may or may not exist depending on whether the policy is an obligation or access-control type policy. Some policies have "signs" where a positive sign indicates permission and a negative sign indicates denial. Table 2.5 summarises the basic (non-composite) policy types of Ponder, their purposes and points of implementation (enforcement).

Authorisation policies are access-control policies that specify what subjects are allowed to do on targets while obligation policies specify what subjects must do on targets when an event occurs. Subjects are users, principals or automated manager-components that have management responsibility, and targets are resources or service providers, which are objects with interfaces. A subject accesses a target by invoking methods visible on the target's interface.

The syntax of access control and obligation policies is shown in Figure 2.7. Note that the subject field is optional in Ponder authorisation-policy specification as shown in Figure 2.7. Similarly, the target field is optional in the Ponder obligation-policy

```
inst  (auth+ | auth-) nameOfPolicy {
      [subject [<type>] domain-scope-expression;]
      target[<type>] domain-scope-expression;
      action action-list;
      [when constraint-expression;]
      }
```

```
inst  (oblig) nameOfPolicy {
      subject [<type>] domain-scope-expression;
      [target[<type>] domain-scope-expression;]
      do obligation-action-list;
      [catch exception-specification;]
      [when constraint-expression;]
      }
```

Figure 2.7: Syntax of Ponder Authorisation and Obligation Policies

specification as shown in Figure 2.7, as subjects may perform actions on themselves when an event occurs.

Ponder has the concept of composite policies, which can be used for simple grouping of policies or to capture an organisational structure as it is. These concepts are suitable for policy-based management. Four composite policy constructs, namely *Group*, *Role*, *Relationship* and *Management Structure* are supported.

A group is used for organising policies by grouping related policies based on their target, the application they apply to or other criteria. A role is a special type of group where all the policies in the construct have the same subject. Given an organisational position, a role is a set of policies (obligation, authorisation, refrain, etc.) whose subjects are the ones who assumed that position. Once a role is defined, it is possible to derive sub roles by inheritance. Roles have a different semantics in the new Ponder. A role is redefined as an atomic construct that has a set of obligation policies.

Relationships are used to relate roles by specifying policies that are part of the interaction between roles. Management Structures are constructs that enable administrators to define a policy that reflects the structure of an organisation. A management-structure policy contains roles, relationships and other management structures.

Deployment Model In [DLSD01], an object-oriented deployment model for Ponder was presented. In this model, shown in Figure 2.8, policies, domains and policy-enforcement agents are conceived as objects. Enforcement agents are objects to whom the actual policy enforcement is delegated. The enforcement of authorisation

policies is delegated to *Access Controllers* whereas that of obligation policies is delegated to special agents called *Policy Management Agents* (PMAs). The deployment model bases the deployment around three services, namely *Policy Service*, *Domain Service* and *Event Service*.

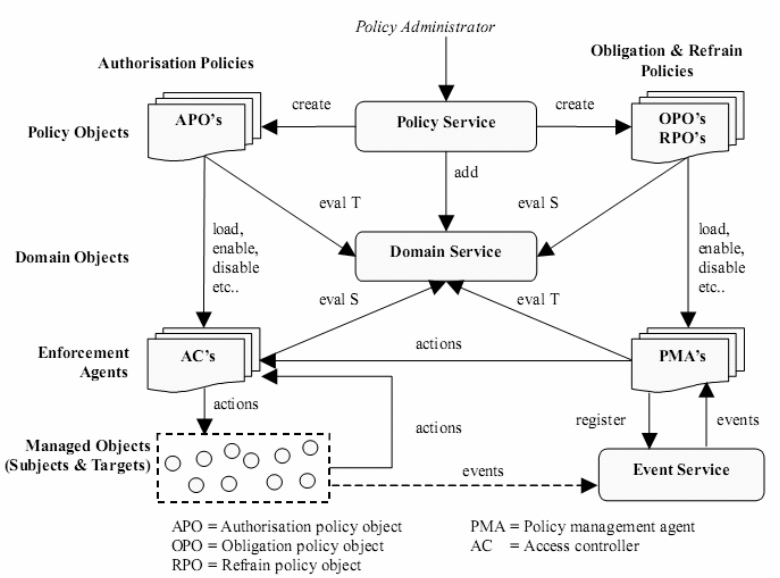


Figure 2.8: The Ponder Deployment Model [DLSD01]

Ponder2

Ponder2 [Pon] is the latest version of the Ponder policy framework. It is a generic object management system that allows dynamically loading, unloading, enabling and disabling of managed objects. Generally a Ponder2 managed object is an active object that is capable of receiving action commands and performing actions. A managed object can be either part of the management system's architecture (e.g., a discovery service) or an adaptor to a managed system (e.g., a sensor managed object serving as an adaptor to a managed sensor). Managed objects are written in Java and a high-level language referred to as PonderTalk is used to specify commands for controlling and interacting with them.

Ponder2 consists of four components, a domain service, an obligation policy interpreter, an authorisation enforcement and a command interpreter. The domain service

is used to store and access managed objects in a structured manner. The obligation policy interpreter interprets event-condition-action rules. The command interpreter accepts and interprets a set of commands directed to managed objects in the domain structure. The authorisation enforcement provides the ability to specify fine grained positive and negative authorisation policies for managed objects. Managed object codes are loaded into the Ponder2 system dynamically resulting in a factory managed object that is comparable to classes in Java or other object oriented languages. Commands can then be sent to this factory managed object to create managed objects which themselves can be sent other commands.

PonderTalk PonderTalk is a Smalltalk-based language used for configuration and control of a Ponder2 system. Consequently, Ponder2 policies are specified using PonderTalk.

A PonderTalk specification is a set of statements separated by a "." that are either assignments (to a temporary variable created on the fly) or a reference to a managed object followed by zero or more commands (messages) sent to the managed object. The latter form of statements specifies message interactions. Message interactions have return values, which are PonderTalk objects, and hence they can be assigned to variables or passed as data to other commands. In the example PonderTalk specification shown in Figure 2.9, line 2 is a PonderTalk statement that assigns the result of the statement $root \quad load : \text{“}Domain\text{”}$ to the temporary variable $domainFactory$. Commands can be cascaded, by using semi colons, and sent to a single managed object. The cascaded commands shown in lines 5 - 7 create factories for domain, event template and authorisation policy objects respectively and store them in the domain $root/factory$. There are three types of commands that can be sent to a managed object: unary messages, binary messages and keyword messages. Unary messages are simple commands sent to a managed object without any data. For example, the command $create$ in the statement $domainFactory \quad create$ in line 5 is a unary message to the $domainFactory$ object. The $domainFactory$ creates a domain managed object, which can store other managed objects including domains in a structured manner. In the $root \quad load : \text{“}Domain\text{”}$ statement, $load : \text{“}Domain\text{”}$ is a keyword message where $load :$ is the keyword and $\text{“}Domain\text{”}$ is the associated data (argument), which is a string object identifying the managed object code to be loaded. Binary messages

```
1 //Import the Domain code.
2 domainFactory := root load: "Domain".
3 //Create a domain called factory to put all the factories including
4 //the domain factory and a domain called policy to put all the policies.
5 root at: "factory" put: domainFactory create;
6     at: "event" put: domainFactory create;
7     at: "policy" put: domainFactory create.
8 root at: "utils" put: domainFactory create.
9 //Put the domain factory into the factory domain, for later use.
10 root/factory at: "domain" put: domainFactory.
```

Figure 2.9: PonderTalk Statements

are operators such as "+" followed by a single object. In all types of messages, the receiving managed object is supposed to understand the meaning of the messages. In PonderTalk, managed objects are referred by their path (including their name) in the domain structure. The *root* domain is the upper most domain and is created by Ponder2 itself.

As we have seen in the previous example PonderTalk statements (Figure 2.9), managed objects not only can receive messages but also can themselves be passed as data in the messages. PonderTalk also has basic (built in) objects that can be passed to managed objects as arguments. These are $Array$, $Number$, $String$, $Boolean$, Nil, $Hash$ and Xml. Arrays are an ordered collection of objects that are created using a PonderTalk statement by putting the elements in a brace preceded by the # symbol, as shown in line 2 of Figure 2.10, or by operations in a Ponder2 managed object (i.e., in the Java code of the managed object). The array shown in this example has various elements: a number, boolean, string and a domain object. Arrays can receive messages. For example, to find out the size of an array we can use the $size$ message as shown in line 3. Hashes are a collection of named objects and are created only by operations in a Ponder2 managed object. For a listing of messages that can be sent to the $Array$ and $Hash$ objects, Appendix A can be consulted.

In addition to the basic PonderTalk objects, there is a special object referred to as $Block$, which behaves much like a function with zero or more arguments. It is created by putting arguments of the block followed by statements, in a square bracket. To execute a block, a special message referred to as $value$ is sent with or without arguments depending on whether the block takes arguments or not. Blocks help parametrise PonderTalk statements. For example, the code for creating and storing a domain in the domain structure shown in lines 5 & 7 of Figure 2.9 can be rewritten

```
1  //An array containing different objects.
2  myArray := #(10 true "http://www.doc.ic.ac.uk" root/utils/newdomain ).
3  arraySize := myArray size.
4  //Block for creating domains. Note that the domain factory stored in the
5  //previous example line 10 of Figure 2.9 is used.
6  domainCreater := (:domainName|
7  root at: domainName put: root/factory/domain create.
8  ).
9  //Store the block object.
10 root/utils at: "newdomain" put: domainCreater.
11 //Using the block object create domains.
12 root/utils/newdomain value: "local"; value: "remote".
13 //Conditional operation using blocks.
14 (myArray size == 4) ifFalse: (root print: "false") ifTrue: (
15 root print: "true").
16 myArray do: (:value| root print: value).
```

Figure 2.10: PonderTalk Objects

using a block as shown in lines 6 - 8 of Figure 2.10 and reused for performing the same task for various domains as shown in line 12 of Figure 2.10.

Blocks can be executed conditionally using the $ifTrue$: and $ifFalse$: messages of PonderTalk's $Boolean$ object. For example, in the statement shown in lines 14 & 15, depending on the result of the message sent to the array object either the first block (which prints out $false$) or the second block (which prints out $true$) is executed. In this specific case, the second block is executed since our array has four objects.

The messages do : and $collect$: enable an iterative execution of a block for each element, with the element as an argument, in an array or hash managed object. While the do : message returns the result of the last execution, the $collect$: message puts the result of each execution in an array object and returns this object. For example, the statement shown in line 16 executes the block for each element in the array resulting in all the elements of the array being printed out. The $whileTrue$: and $whileFalse$: messages enable looping the block as long as its return value is true or false respectively.

Ponder2 Events and Policies Events and policies are specified using PonderTalk and their creation is facilitated by event and policy factory objects, namely $Event\ Template$, $ObligationPolicy$ and $AuthorisationPolicy$, which are provided by Ponder2. These factories are created and stored in the domain structure as shown in Figure 2.11.

```
1 //Import and put event and policy factories.
2 root/factory
3     at: "event" put: ( root load: "EventTemplate" );
4     at: "ecapolicy" put: ( root load: "ObligationPolicy" );
5     at: "authpolicy" put: ( root load: "AuthorisationPolicy" ).
```

Figure 2.11: Ponder2 Event and Policy Templates

event-factory create : *arguments-array*

Figure 2.12: Syntax of a Ponder2 Event Type

Ponder2 events are a set of name-value pairs. The set containing the name list is referred to as an event template. Each event has an event template (type), which is created by using a *create* message with the list of argument names to the event template factory as shown in Figure 2.12. Once the event template is created, an event of that type can be created by sending messages, with or without values, to the template. After the event is created, it is propagated to all policies that have subscribed to that event type. Figure 2.13 shows example Ponder2 event types and events. Events can also be created by managed objects (i.e., in the Java code of the managed object) provided that they are given the event template for the events they are expected to generate and they understand the semantics of the event types (i.e., the arguments). The event templates can be passed to the managed objects through messages.

Obligation policies are created by using the obligation policy factory object and setting the event type, condition and actions of the policy. The event type, condition and actions are set through a message sent to the policy object. The condition and actions are specified using blocks as shown in Figure 2.14. When an event of the specified type occurs, the policy evaluates the condition block and if the result is true, it executes the action block. The blocks get the values for their arguments from the

```
1 //Create carArrived and carLeft event types.
2 event := root/factory/event create: #("type" "colour" "numberPlate")
3 root/event at: "carArrived" put: event.
4 event := root/factory/event create: #("numberPlate")
5 root/event at: "carLeft" put:event.
6 //Create and send carArrived and carLeft events.
7 root/event/carArrived create: #("sedan" "green"  "PONDER2CAR").
8 root/event/carLeft create: #("NONPONDER2CAR").
```

Figure 2.13: Example Ponder2 Events

```
policy := root/factory/ecapolicy   create.
policy   event : event-type;
condition : [: arg | boolean-expression];
action : [: arg | statements]
```

Figure 2.14: Syntax of a Ponder2 Obligation Policy

```
policy := (root/factory/authpolicy
subject : subject-path-and-name
action : action-name
target : target-path-and-name
focus : focus-type).
[policy   mode.]
[policy   condition.]
```

Figure 2.15: Syntax of a Ponder2 Authorisation Policy

event.

Authorisation policies are created by using the authorisation policy factory object and specifying the subject, action, target and focus of the policy, as shown in Figure 2.15. An authorisation policy can protect either or both the source (subject) and target from a given action; which one it is protecting is specified by the focus type using the values t for target and s for subject. The source can be protected from performing actions or accepting replies to actions that are harmful to itself or other subjects in the domain. The target can be protected from unauthorised subjects trying to perform an action (access control) or from sending back the result of an action that contains sensitive information (privacy control). The Ponder2 system can be configured to allow or deny all actions on all managed object by default and then interpret the explicit specification indicating the subset of actions that are denied or allowed respectively. After a policy is created, its mode, i.e., whether it is a negative (deny) or positive (allow) authorisation, can be set by sending the optional (shown in square braces in the figure) mode type message to the policy. The default mode of a Ponder2 authorisation policy is positive; it can be set to negative by sending the $reqneg$ or $repneg$ messages to the policy to deny invocation (request) and reply respectively. Conditions of an authorisation policy, which are optional, can be specified using the arguments of the operation that the policy is protecting. Figure 2.16 shows example Ponder2 obligation and authorisation policies.

Ponder2 supports both explicit specification of authorisation and obligation policies, which are necessary for our framework. In addition, the Ponder2 framework has a

```
1  //An obligation policy that prints out the details of a car, and stores
2  //it (the corresponding managed object) in the cars domain using its
3  //number plate as the managed object name when a car arrives at a
4  //parking station if the car is not of type SUV.
5  policy := root/factory/ecapolicy create.
6  policy event: root/event/carArrived;
7  condition: (:type | type != "SUV");
8  action: ( :type :colour |
9  root print: "A " + colour + " " + type +" car is entering the station.".
10 root/carpark/cars at:
11     numberPlate put:(root/factory/carmanagedobject create).
12 ).
13 //An authorisation policy that prevents cars from extending the
14 //parking period.
15 policy := (root/factory/authpolicy
16 subject: root/carpark/cars
17 action: "extendPeriod:"
18 target: root/carpark/meter
19 focus: "t").
20 policy reqneg.
```

Figure 2.16: Example Ponder2 Policies

number of advantages that make it suitable for a wider range of application domains. Its design is simple in that it has only few built-in elements. It is extensible in that one can extend it with new functionality, to interact with new services and to manage new resources. It is self-contained in that it provides a means to interpret policies, enforce policies on managed resources and interact with the policy system. It is also scalable in that it can be deployed on systems with constrained resources.

Law-governed Interaction

Law-governed interaction (LGI) [MU00] is a framework that deals with the specification, deployment and enforcement of coordination policies for heterogeneous distributed systems. Given a group of agents (agents, in LGI, are systems involved in the interaction, including human beings), LGI ensures that the interaction among this group, referred to as $\mathcal{L}$-group, complies with the law ($\mathcal{L}$) of the group. An $\mathcal{L}$-group $\mathcal{G}$ is defined as a four-tuple $\langle \mathcal{L}, \mathcal{A}, \mathcal{CS}, \mathcal{M} \rangle$ where $\mathcal{L}$ is an explicit set of rules, $\mathcal{A}$ is the set of agents that are members of the group, $\mathcal{CS}$ is a set of control states, one per member of the group and $\mathcal{M}$ is the set of messages, referred to as $\mathcal{L}$-messages, which can be exchanged between members under the law.

Specification An LGI law is an event-condition-action rule defined over a subset of events, referred to as regulated events, occurring at members of an $\mathcal{L}$-group. Events trigger the evaluation of the law resulting in operations referred to as the ruling of the law. The law is specified using a restricted version of Prolog (goals such as $retract$ and $call$ are not permitted) that has two additional types of goals called a do-goal and a sensor-goal. A do-goal, which has the form $do(p)$, where p is a primitive operation appends the term p to the ruling of the law. A sensor-goal, which has the form $t@CS$, where t is a Prolog term unifies term t with each term in the control state of the agent. When an event occurs at an agent x, the law is evaluated in the context of the control state of x ($\mathcal{CS}_x$) resulting in a list of primitive operations, which are the ruling of the law for this event. The operations can be performed on the control state of the agent (e.g., incrementing the value of a parameter in the agent's control state or adding a term to the control state), on messages (e.g., forwarding a message) or agents (e.g., imposition of an obligation on an agent). These operations are immediately carried out possibly resulting in more events and more rulings. Figure 2.17 shows a law for regulating membership in a peer-to-peer medical community. The preamble of the law, which specifies the initial setting of the community, states that the community accepts certificates from two certifying authorities, namely $ca1$ and $ca2$ and in the beginning all agents have empty control state. The rest of the rules are explained in the figure.

Deployment Model In LGI, three entities, namely the controller, controller-server and the secretary facilitate the deployment and enforcement of the law. The law of a group is enforced by trusted agents referred to as controllers. Every member of the group has a controller through which every $\mathcal{L}$-message between the member and other members passes, as shown in Figure 2.18. The controller of an agent also keeps the control state of the agent and deals with computation of the ruling of the law for every event at that agent. A controller-server is a name-server that keeps track of active controllers. An agent and its controller can be hosted on the same machine, but in the most general case the agent can request for a controller from a controller-server when it wishes to engage in an LGI group. LGI supports two types of groups – an explicit group where membership is regulated and maintained and an implicit group where any agent operating under the law can be a member and there is no membership maintenance. In an explicit group, the secretary is responsible for

Preamble:
$authority(ca1, CA1PublicKey).authority(ca2, CA2PublicKey).initialCS([]).$
Rule 1:
$certified(X, cert(issuer(ca1), subj(X), attr([found(X)])))$
$: -do(+member).$
The agent is accepted into the community without requiring approval from already admitted members if it can present a certificate from ca1 stating that it is a founder.
Rule 2:
$certified(X, cert(issuer(ca2), subj(X), attr([md(X)])))$
$: -do(+certified).$
The agent must establish that it is representing a doctor by presenting a certificate from ca2 with the attribute MD. If the agent establishes that, the goal $do(+certified)$ *will result in the addition of the term* $certified$ *to the agent's control state.*
Rule 3:
$sent(X, allow, Y) : -member@CS,$
$if\ (friend(Y)@CS)\ then\ do(deliver(X, alreadyAccept, X))$
$else\ do(+friend(Y)), do(forward(X, allow, Y)).$
If the agent attempts to send a message approving admittance of another agent, forward the message if the agent (a) has membership, and (b) has not previously given Y permission to join.

Figure 2.17: An Example LGI Law [IMN04]

maintaining the law of the group, the list of members and initial members, and the initial control state that is assigned to a new member. The initial members are roles defined in the law and can be played by any of the agents that requests to fill that role.

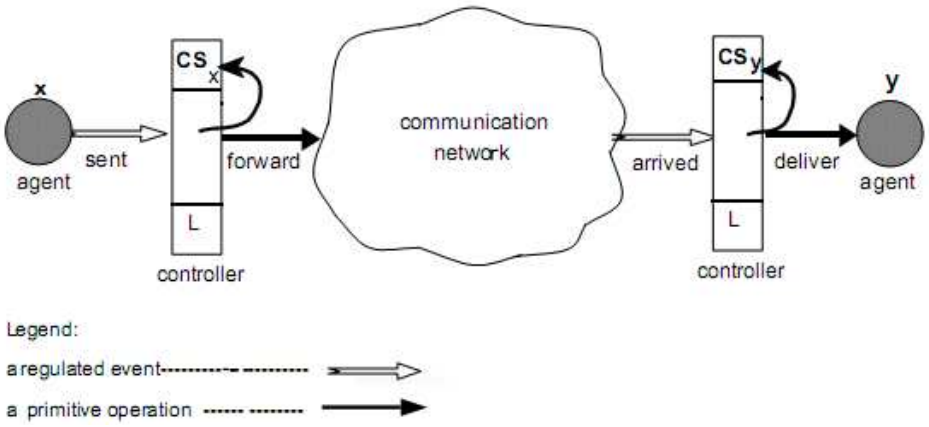


Figure 2.18: LGI Law Enforcement [IMN04]

Once the agent gets its controller, it can join a group by sending a join request to the secretary of the group by identifying its controller and the role it wants to play if it is going to play one of the roles. The secretary then provides the law and the initial control state of the agent to the controller of the agent. If the agent is playing one

of the roles, its control state contains a term stating its role. The secretary might perform authentication of both the agent (through a password) and the controller (through a certificate) before admitting an agent to the group, if the law of the group so requires.

LGI is a powerful approach for managing interactions among distributed agents. It, however, has two limitations in relation to our objective. The first one is that since it focuses on interactions (i.e., authorisations in LGI parlance), it has a minimal support for obligation policies that are needed to perform adaptation. The support for obligation policies is based on its *enforced obligation* concept, which can initiate interaction but is not allowed to change the control state thereby limiting the ability of obligation policies to trigger adaptation by changing the behaviour of the autonomous system. The second limitation is caused by its need for a controller entity to enforce the law. In applications where there are trusted agents that can serve as controllers, LGI is appealing since agents can exchange their LGI messages via these controllers. However, in teams of mobile autonomous systems that are formed and dissolved in a relatively short period of time, it is infeasible to assume the presence of trusted controllers. Although LGI allows putting a controller in the same system as the agent, this compromises the protocol as the interaction will be performed without a trusted controller.

Policy-based Configuration management

Cfengine [Bur93, Bur95] is a policy-based configuration management framework for systems and network administration. It enables automation of administrative tasks such as installation, configuration, updating and maintenance of networked hosts. Hosts are divided into groups called classes based on their attributes, and policies are specified for each class. Since hosts know their attributes, they can determine to which class they belong to and pick policies that apply to themselves and behave accordingly.

Specification A Cfengine configuration specification, shown in Figure 2.19, consists of three main sections: groups, control and policies. In the groups section, classes are defined from groups of hosts. The control section is used to set internal

```
groups :
    myclass = (host1  host2  host3)
control :
    actionsequence = (links  files)
links :
    class1.class2.class3 ::
        /bin/tcsh  ->  /usr/local/bin/tcsh
files :
    myClass ::
        /usr/local  owner = root  mode = o-w  action = fixall
    (linux|solaris).Hr12.onTheHour.!exception_host ::
        /etc/passwd  mode = 0644  action = fixall  inform = true
```

Figure 2.19: An Example Cfengine Configuration Specification

variables, the most important one being the state of policies (i.e., whether they are active or not). After these two sections, follow the policies. A Cfengine policy is an event-condition-action rule that has two elements, namely class and statement. The class, in its most general form, is used to encode both the event and condition of the policy. The statement part of the policy is used to specify the actions. Policies are grouped into sections, which are effectively names for the policy groups and serve the purpose of identifying the policy group for activation or deactivation.

In the example Cfengine configuration specification [Bur95] shown in Figure 2.19, a class called myClass is defined from three hosts (in the groups section) and a policy for this class is specified (in the files section). A host interpreting this specification will perform the action in this policy if it belongs to myClass, i.e., if it is either host1 or host2 or host3. If a policy applies to all hosts then the class specifier can be omitted. Classes can also be specified by using expressions as shown in the second policy of the files section. This expression specifies (1) that a host is a member of this class, if its operating system is either *Linux* or *Solaris* and if it does not belong to a class labelled as *exception_host*, (2) a trigger (event), which specifies that the policy is triggered *on the hour* and (3) a condition, which specifies that the time interval must be between 12:00am and 12:59am.

Deployment Model Cfengine's deployment model is based on an agent, namely the Cfengine agent, which runs on every host and enforces policies, and a collection of helper services used for remote operations and public-key based authentication associated with these operations. Later versions of Cfengine are enhanced by the addition of new components such as a learning component for anomaly detection.

Policies are stored in a central repository accessible by all hosts in the administrative domain from which they are fetched upon startup.

Although Cfengine is able to specify adaptation policies, the events it can respond to (and hence the context) is limited by its very nature of class evaluation approach. Classes, which are used to encode events, can be evaluated either as a result of the characteristics of the system on which Cfengine is running (e.g., operating system, time of day on the system, etc.), or by making the system a member of a named group or by using an explicitly defined identifier. These types of events, which are sufficient to specify adaptation in systems administration (for which Cfengine is targeted for), are not sufficient for our purpose where the context (and hence the domain of associated events used to trigger policies) needs to be larger. In addition, Cfengine does not have a means to explicitly specify authorisation policies, which are necessary in a team to selectively allow access to resources for some team members.

Artificial-intelligence Based Policy Framework for Autonomic Systems

In [KW04], a policy framework based on the concepts of artificial intelligence was proposed. In this framework, an autonomic system was modelled as a rational agent [NR03] that perceives and acts upon its environment by selecting actions that are expected to maximise its objective. The actions are selected on the basis of information from sensors and built-in knowledge.

The authors demonstrated how three of the notions of a rational agent can be incorporated into policies for autonomic computing. The three notions are *reflexive agenthood*, *goal-based agenthood* and *utility-based agenthood*.

Reflexive agents use if-condition-then-action rules. The rationality in these agents is that they select actions appropriate for the condition, assuming that the designer who encoded the rules has encoded them rationally. Goal-based agents select actions that could enable them to achieve specified goals. The rationality in these agents is that they select actions that lead them to a desired state. Utility-based agents select actions so as to maximise their utility function. The rationality in these agents is that they select actions so that the actions result in maximum utility.

These notions of rational agents were captured in three types of policies called action

policies, goal policies and utility-function policies.

Action policies specify the actions that must be taken whenever the system is in a given current state. These policies have the form *IF (Condition) THEN (Action)*, where *Condition* is either a specific state or set of states that satisfy the given condition. Some important points about action policies are listed below.

- They do not require an operational model of the system.
- They require the user to have a larger amount of domain expertise since the user is required to know the control variables and sets of states (state space).
- They are susceptible to conflicts.

Goal policies specify either a single desired state or one or more criteria that characterises a set of desired states. Some important points about goal policies are listed below.

- They give the system more autonomy by allowing it to choose the best actions under the current conditions.
- They require lesser domain expertise from the user, compared to action policies, as they are expressed only in terms of the state space.
- Similar to action policies they are susceptible to conflicts.
- They need system models and planning algorithms.

Utility-function policies express the value associated with each state instead of classifying the states as desirable and not-desirable, which goal policies do. The assignment of real-valued desirability value to each state enables a finer distinction between states. For example, if a certain goal policy is not able to take the system to a desirable state, it is considered failed whereas a utility policy can take the system to a state that has a higher utility than the current state even if it may not be able to achieve the maximum utility. Some important points about utility-function policies are listed below.

- They inherently avoid conflicts. Unless they are mixed with action or goal policies in the same component within a system, conflict does not arise in utility policies.

- They need models and optimisation algorithms. But once these are designed the process is straightforward.
- It is difficult to specify utility functions.

In this section, we have discussed different policy-based systems and assessed their applicability to our system. We note that two of the approaches, the IETF policy framework and Cfengine [Bur95], lack authorisation policies. Cfengine also has a limited domain of events due to the approach it uses to encode events. LGI's [MU00] use of a trusted controller entity to exchange law governed messages makes it less suitable to dynamic teams. In addition, due to restrictions on operations that can be performed on the control state, its concept of *enforced obligations* is not as powerful as other approaches such as Ponder2. Ponder2, on the other hand, supports explicit specification of both authorisation and obligation policies with a self-contained deployment model that is suitable for dynamic teams. These and other qualities such as extensibility, scalability and ease of use make it appealing to our system.

In the previous section, we have seen systems such as robot control software, which enable robots to manage themselves without human intervention. The concept of self-management, however, is not limited to robots and agents. The autonomic computing paradigm employs this concept to enable self-management in a wide range of application domains. In the next section, we present an overview of this paradigm with representative autonomic computing architectures.

2.7 Autonomic Computing

Autonomic computing [Hor01, KC03] is a paradigm inspired by the Autonomic Nervous System (ANS) [Bri], which regulates organ functions mostly without any conscious effort by the organism. Much like the ANS, an autonomic computing system can perform self-management without the administrator's intervention, provided that it is initially given high-level objectives from the administrator. Autonomic computing is believed to play a key role in overcoming the difficulty of managing today's complex computing systems.

The primary goal of autonomic systems is self-management, which as outlined in

[KC03], has four aspects, namely self-configuration, self-optimisation, self-healing and self-protection.

1. Self-configuration: the ability of a system to automatically configure itself in accordance with a declarative specification such as high-level policies that specify the desired behaviour of a system. When a new entity joins a self-configuring system, a mutual adaptation, i.e., by both the new entity and the system, takes place leading to a seamless composition.

2. Self-optimisation: the ability of a system to continually monitor its efficiency and adapt the system parameters or components in order to increase efficiency based on specified measures of utility such as performance, cost, etc.

3. Self-healing: the ability of a system to detect, diagnose and repair problems resulting from bugs or failures in software and hardware.

4. Self-protection: the ability of a system to protect itself from malicious attacks as well as problems caused by entities outside the system but propagated to the system, and forecast problems based on reports in order to proactively mitigate them.

In addition, an autonomic system is required to be aware of itself (its states and behaviours) and current context. It should also be built on standard and open protocols so that it operates in a heterogeneous environment, and be anticipatory so that it anticipates its needs, behaviours and the environment and manages itself proactively. In [WPT03], it is illustrated that proactive computing complements autonomic computing.

Research in autonomic computing has resulted in a plethora of architectures that have incorporated one or more of the autonomic system properties targeted for specific applications (e.g., network, storage, etc. management). Parashar et al. [PH05] present a good summary of these application-specific autonomic architectures.

In this section, we consider related work that address most of the issues of self-management using the autonomic computing paradigm for a broader application domain.

The IBM Autonomic Computing Architecture

White et al. use autonomic elements [KC03] as building blocks of an autonomic system and propose an approach by stating the required behaviours and interfaces of autonomic elements. An *autonomic element* is a component comprising two entities, namely an autonomic manager and a managed element. The basic structure of an autonomic element is shown in Figure 2.20. The autonomic manager monitors the managed element and its environment, analyses, plans and executes the plan on the managed element. An autonomic element is responsible for managing its own behaviour, and interaction with other autonomic elements in accordance with policies.

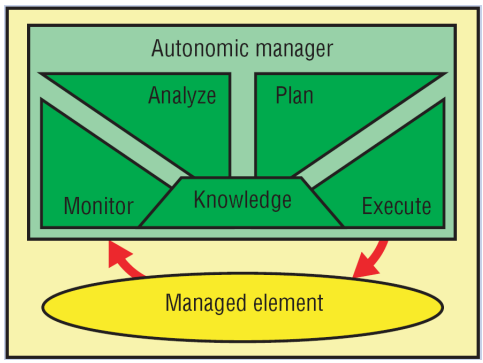


Figure 2.20: The Autonomic Element [KC03]

Two fundamental goals an autonomic computing architecture should accomplish have been put forward:

1. It must describe the external interfaces and behaviours required to make an individual component autonomic.
2. It must describe how to compose systems out of these autonomic components in such a way that the resulting system is also autonomic.

The behaviours of an autonomic element can be classified as required and recommended, where the required behaviours are:

1. An autonomic element must be self-managing (self-configuring, self-healing over internal failures, optimising its own behaviour, protecting itself from external probing and attack).
2. It must be capable of establishing and maintaining relationships with other autonomic elements.

3. It must manage its behaviour and relationships so as to meet its obligations.

and the recommended behaviours are:

1. It should ask for a realistic set of requirements when requesting a service from another element.
2. It should offer a range of performance, reliability, availability and security associated with its service.
3. It should be able to translate requirements for its service characteristics into requirements for any services that it needs to request from other elements.

Autonomic systems are formed from autonomic elements with the help of special autonomic elements that implement system level behaviours, referred to as infrastructure elements:

1. Registry : used for service discovery.
2. Sentinel: used for monitoring services to other elements.
3. Aggregator: combines two or more elements and uses them to provide improved service.
4. Broker: facilitates interaction.

Policies have been identified as a means for specifying desired behaviours of autonomic systems, and policy-based self-management as the focus for the use of policies in autonomic computing. The proposed policy-based management approach was based on the Artificial-intelligence (AI) based policy framework discussed in Section 2.6.1.

Unity

Tesauro et al. present an autonomic computing architecture called Unity [TCW+04, CSWW04], which is based on multiple interacting agents and realises a number of autonomic system behaviours including self-configuration, self-healing and self-optimisation. They decompose the autonomic computing problem into decentralised

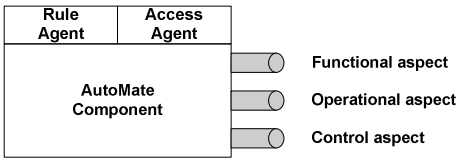


Figure 2.21: A Conceptual View of an AutoMate Component

autonomous agents using an agent-based system approach [Jen00] and model these agents using the autonomic element architecture presented in [KC03]. All components are modelled as autonomic elements; this includes computing resources such as servers, management entities such as arbiters and infrastructure elements such as policy repositories and registries. Unity aspires to create a dynamic multi-application environment that can cater for self-assembly of systems to achieve an application's goal and optimal allocation of resources to the logically separate applications, which use the shared finite resources. The main elements in Unity are registry, arbiter, policy repository and sentinel. Resources such as servers announce their availability and applications announce their interest for resources, to the arbiter, which uses utility functions to deal with optimal allocation of resources to applications. Elements use the registry to locate other elements including the policy repository, which contains policies governing the role of each element. Upon startup, elements retrieve policies related to them from the policy repository. The authors have implemented a prototype of the architecture for a data centre application that provides computation resources to run multiple applications for many users.

AutoMate

AutoMate is a framework that extends the Open Grid Service Architecture (OGSA) [TCF+02] to enable autonomic computing for Grid applications. It has three layers, from bottom to top, namely System, Component, Application, and modules common to all layers called Engines. The system layer is built upon the OGSA and provides peer-to-peer messaging and event services. The component layer deals with the definition, execution and discovery of components. The application layer deals with autonomic composition of components to meet application requirements. AutoMate also has three engines, namely Trust/Access control, Deductive and Context-awareness, which are decentralised networks of agents.

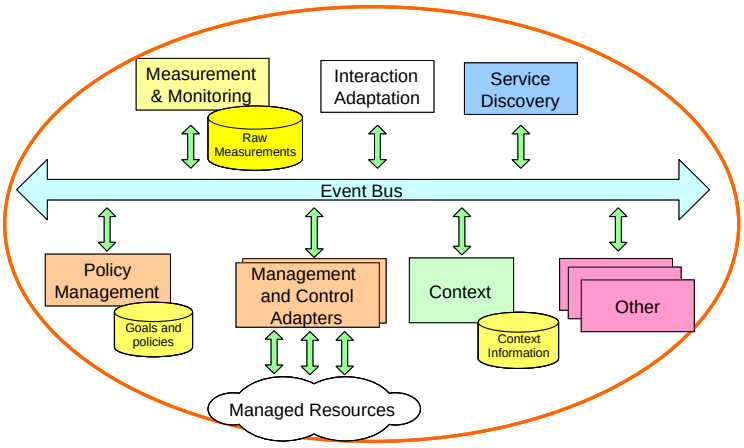


Figure 2.22: Architecture of an SMC

The key autonomic entities in AutoMate are components that model an autonomic element as a computational element managed by rule and access agents as shown in Figure 2.21. An AutoMate component exports its computation behaviours (functional aspect), operational behaviours such as computational complexity and resource requirement (operational aspect), and the adaptation and management provisions (control aspect). The functional behaviours are used by the composition module to select appropriate components for applications while the operational behaviours are used to optimise component selections. The components contain access policies and rules, which are used by the access and rule agents to manage the security and adaptation of the component respectively. The rule agent is part of the deductive engine, which enables execution and dynamic change of rules that in turn change the behaviour of the component.

The Self-managed Cell Architecture

A Self-managed Cell (SMC) [LDS+08] is a closed-loop systematic organisation of management services that represent a set of hardware and application components in a single device, or multiple devices collaborating to achieve a common goal. The management services interact with each other through an asynchronous event bus. An SMC contains measurement and monitoring, event, discovery, context and policy services, which are required in order to implement a feed-back loop adaptation mechanism. The architectural pattern of an SMC is shown in Figure 2.22.

The function of each core service is briefly described as follows. In order to form its

closed loop adaptation, an SMC should at least have an event and policy services. The remaining core and other services can be added as per the application's need. In essence, the SMC is an architectural pattern that gives the freedom to design a concrete architecture befitting an application domain. The *Policy Service* is the means of specifying the adaptive behaviour of an SMC. The *Event Bus* is responsible for asynchronous notification of events to different management services of the SMC. Event notification is a crucial element of an SMC because adaptation, protection and other self-management actions are specified in terms of obligation policies, which are triggered by events. An obligation policy may perform an action that modifies the behaviour of a single component of an SMC or it may enable or disable other policies to change the overall behaviour of the SMC. The use of an event bus also supports loose coupling between the various SMC services, which can react concurrently and independently to event notifications. However, all communication is not constrained to be over the event bus.

The *Discovery Service* discovers components that are in range and capable of being members of the SMC. The *Measurement and Monitoring Service* is responsible for keeping track of the SMC's operation and generates events when actions need to be taken. Events may indicate situations such as degradation or failure of components. The *Context Service* uses sensors to determine information such as current location, activity and what other SMCs or other entities are in the vicinity as the SMC behaviour will adapt to current context.

A group of SMCs may compose or federate to form a single SMC; this is the approach in the SMC architecture to form a self-managing system from self-managing elements. The SMC architecture incorporates all the functionalities needed for composition (or federation) in each SMC and hence alleviates the need for infrastructure elements. The use of infrastructure elements introduces centralisation, which makes the system prone to single points of failure and reduces the scalability and performance of the system.

2.8 Summary

In this chapter, we have presented background work in robot-software architecture. We have reviewed related work in multi-robot and multi-agent systems that support mission specification and found out that mission specification approaches that explicitly specify behaviours in terms of rules are more adaptive than those that do not use rules. We have also considered related work in capability description and matching. We observed that the UPnP approach can be adapted to suit our requirement. Also, with respect to capability matching we have noticed that structured representation facilitates matching, and that in addition to exact matches, the matching approaches consider other types of matching such as subsumption based ones. We have presented related work on distributed systems deployment, as well as multi-robot and multi-agent architectures that support dynamic role/task allocation. We noticed that approaches that use planning or constraint satisfaction based allocation perform the search for a valid assignment in a large search space and tend to take longer time. We also noticed that in most approaches utility functions are widely used to measure the fitness of a system against the requirement of tasks/roles. We have presented related work on policy-based systems and observed that Ponder2 is preferable for our system. Finally, we have presented an overview of the autonomic computing paradigm and discussed some representative architectures that use this approach.

Chapter 3

Mission Management

3.1 Introduction

In this chapter, we will present an overview of the self-management architecture and then focus on mission management, which is the first layer of the architecture. The components of the mission management system, i.e., policies, roles and missions are discussed, and a role-based approach to mission specification is presented.

3.2 Overview of the Self-management Framework

In real-life applications, multiple autonomous systems deployed on a mission collaborate to use services or resources from another system. To be able to form and use a dynamic collaborative team of autonomous systems in accordance with a high-level mission specification, a means for describing the capabilities of autonomous systems, discovering available heterogeneous autonomous-systems, securely admitting them to the team, assigning them to an appropriate role based on their capability and maintaining the team is necessary. We achieve adaptive management of autonomous systems by employing a self-management architecture that uses policy-based techniques to allow dynamic modification of the management strategy relating to resources, task behaviour, communications and team management, without reloading the basic software within the system. The self-management architecture

comprises three layers, namely mission, team and communication. Each layer is a local component of a distributed control system of that layer, i.e., each autonomous system is loaded with the three-layer self-management architecture. We use Unmanned Autonomous Vehicles (UAVs) as a testbed for our framework. In this and the rest of the chapters, we will use the term UAVs instead of mobile autonomous systems. However, except for the communication link maintenance entity in the communication layer, which performs motion control actions on UAVs, the rest of the architecture does not assume any property specific to UAVs and is applicable to the more general domain of collaborating mobile systems such as disaster relief teams or groups of foot soldiers on a mission.

An overview of the self-management framework is shown Figure 3.1, with each UAV running the self-management architecture (discussed later), and with the high-level properties of the framework indicated in processes and interactions (1) to (12). A mission for a team of UAVs is specified, by a mission administrator, in terms of roles. The mission specification defines how UAVs will be assigned to perform specific roles within the team, based on their capabilities and credentials, as well as when and how to adapt the mission to changes in context or failures. The administrator specifies the mission with the possibility of reusing or modifying policies, roles and mission specifications that are previously defined and stored in a repository (1-2). The repository can be implemented within a mobile node with sufficient resources – on the commander or another UAV. The specification will then be loaded onto a UAV that has the necessary capabilities for managing the mission and consequently the UAV assumes the *Commander* (manager) role (3).

Based on the mission specification, the *Commander* UAV may either preemptively fetch the necessary policies, tasks and roles (i.e., task and role classes) for all future team members (4) or let each member deal with its own fetching (8 & 12 shown in broken lines to differentiate these optional interactions from the necessary ones). The *Commander* UAV then uses its discovery service to discover and authenticate UAVs. The discovered UAVs provide their capability descriptions to the *Commander*, which checks them against the capability requirements of available roles and performs optimal assignments (5-7). The assigned UAVs recursively repeat this process if their mission includes managing other UAVs (9-11), as a result creating a tree that has the *Commander* UAV as a root. This tree is used to communicate management

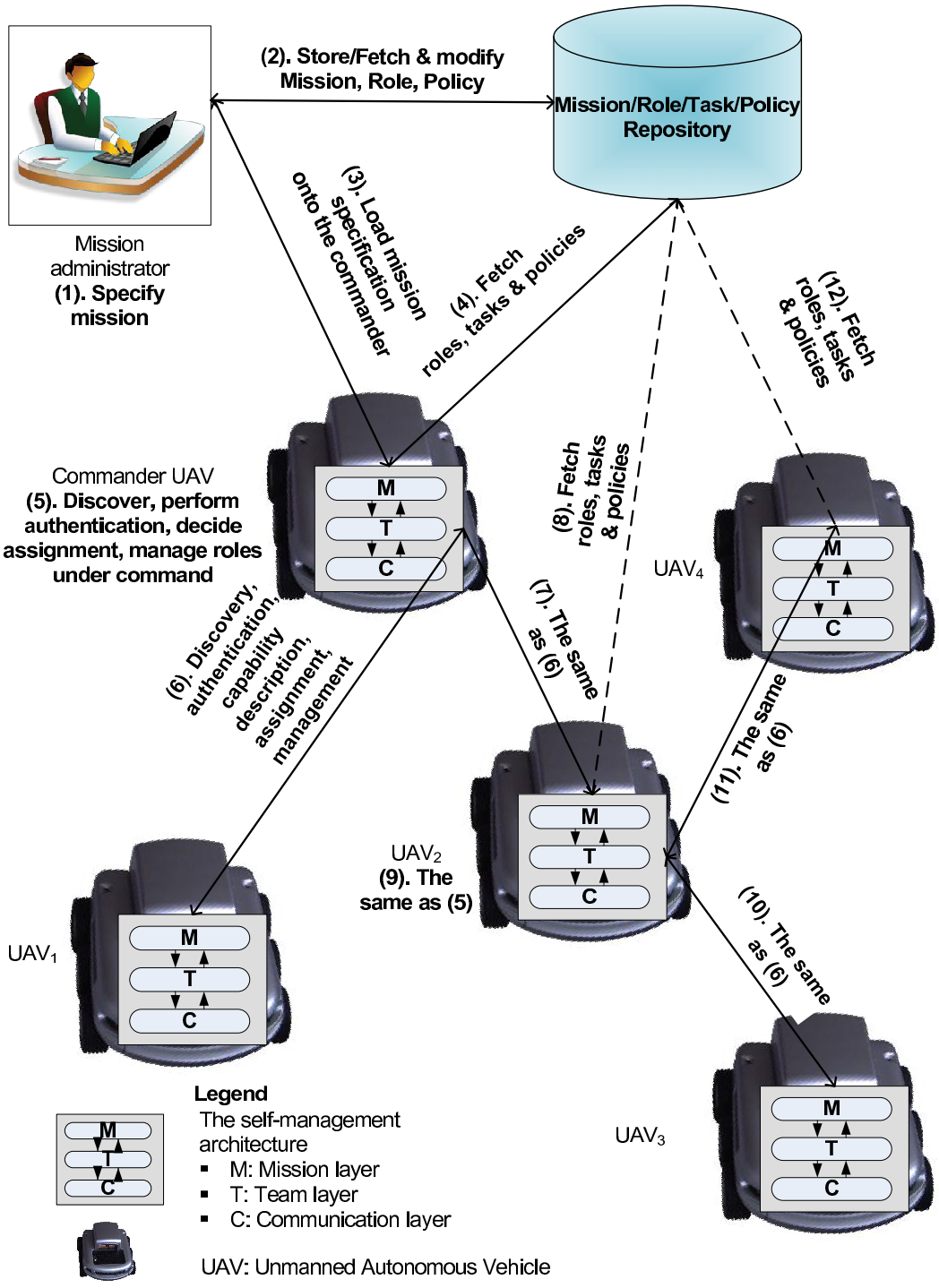


Figure 3.1: Overview of the Self-management Framework

messages, collect state information and organise the roles hierarchically, each with a unique identity so as to make the team robust and capable of recovering from failure.

3.3 The Self-management Architecture

The management framework is designed as a composition of interacting entities organised in three layers, i.e., mission, team and communication management. The architecture is shown in Figure 3.2. To simplify the diagram, not all possible interactions between the entities of the framework are shown. Figure 3.2 illustrates the layers and interactions among them (the autonomous system's control interface is shown later on). We call a role (or a UAV enacting the role) that is responsible for assigning one or more roles a *manager role*, and the roles managed by it *managed roles*

The mission layer consists of three main entities, namely *Mission*, *Role Manager*[1] and *Role*. The *Mission* entity deals with interpreting the mission specification and instantiating the mission using the instantiation specification. The *Role Manager* entity is responsible for loading and withdrawing a role during mission startup and reconfiguration respectively. It also takes control of the UAV and manages the switching between roles when multiple roles are enacted by the UAV. The *Role* entities are the set of roles that the UAV can perform.

The team layer has entities that are available in all UAVs irrespective of the role they are enacting: *Capability Manager*, *State Aggregator* and *Management Tree Node* as well as other entities whose existence depend on whether the UAV is enacting a manager role or not: *Discovery* and *Optimiser*. The management tree is a distributed data structure used to maintain a UAV team. The *State Aggregator* entity is responsible for collecting state information from the managed roles and providing the information to the *Management Tree Node* as well as sending state updates to the manager role. It is also responsible for generating the appropriate events such as communication link disconnection and permanent UAV or link failure when the state information is not received within the specified timeouts. The *Discovery* entity is responsible for discovering UAVs and passing the information to the *Optimiser* entity, which makes the assignment decision. The *Capability Manager* entity is responsible for generating and communicating the capability description of a UAV.

[1] It is worth noting that the Role Manager (written in title case, throughout the thesis, to avoid confusion) is an architectural element found in the mission layer while a manager role is any role that performs role assignments, i.e., manages other roles. Also, note that a manager role is not the same as the commander role. A team has one commander role (at the root of the hierarchy) but many manager roles (all non-leaf roles in the hierarchy are manager roles). A manager role effectively performs the same task for its sub-team as the commander role does for the whole team.

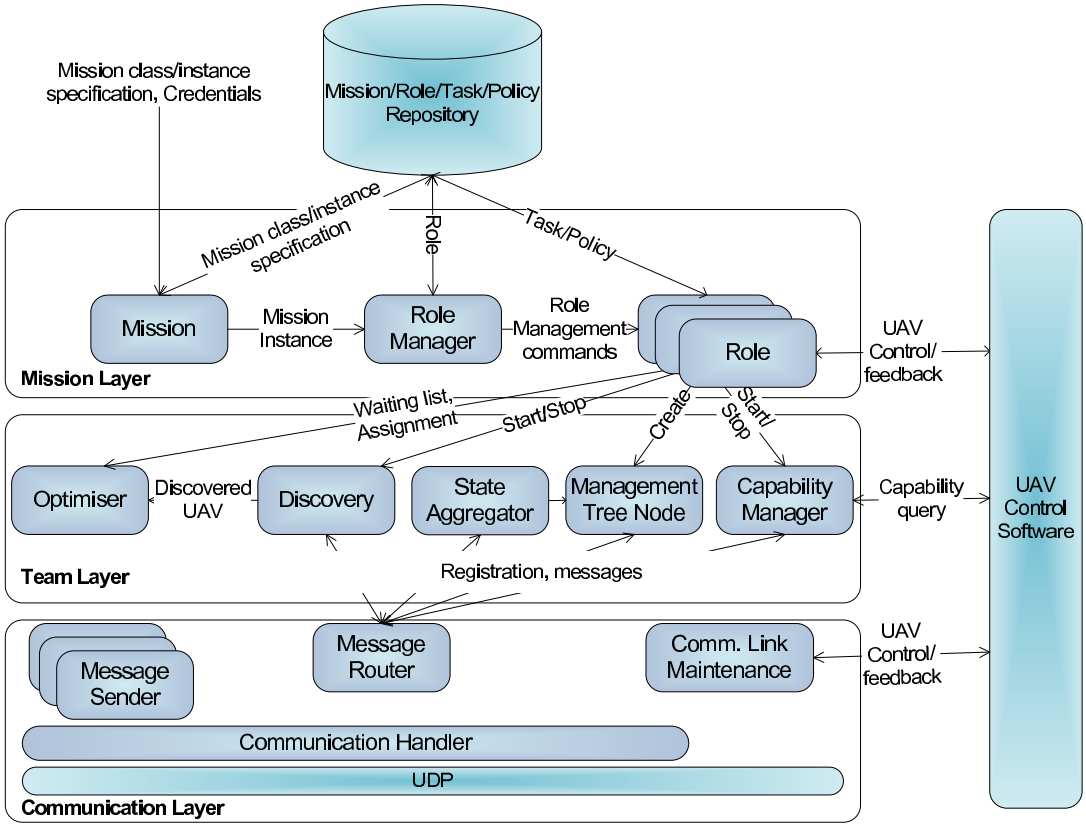


Figure 3.2: Self-management Architecture

The communication layer has three main entities, namely *Message Sender*, *Message Router* and *Communication Link Maintenance*. A connectionless communication mechanism is used for most of the messages exchanged in our system and hence these entities are built upon UDP. However, although connectionless communication is sufficient for most of the messages exchanged between UAVs, reliable delivery is necessary for critical messages such as role assignment and capability description. This service is provided by the *Message Sender* entity. The *Communication Handler* entity provides a secure communication channel between UAVs. The *Message Router* entity forwards incoming messages to the relevant roles and other entities in the UAV. The *Communication Link Maintenance* entity deals with potential communication link failures by means of preventing failures and facilitating the provision of future intermittent communication links should failures are bound to happen.

The management architecture stores all architectural entities as well as tasks, policies, roles and other entities in a domain structure in order to facilitate policy enforcement. In order to allow for policies that perform remote operations, the framework

supports policy-based importation of remote role references and storing them in the domain structure. Consequently, the domain structure is also a means of maintaining a collaboration organisational structure which in general may have a different organisational structure than the hierarchical management structure.

The UAV control software interface is shown in Figure 3.3. The control interface of robots varies depending on the type of the robot. While some provide complex tasks such as mapping and localisation, others provide basic motion and sensory interfaces. The Koala robot [kt], which was used as a testbed for the framework,

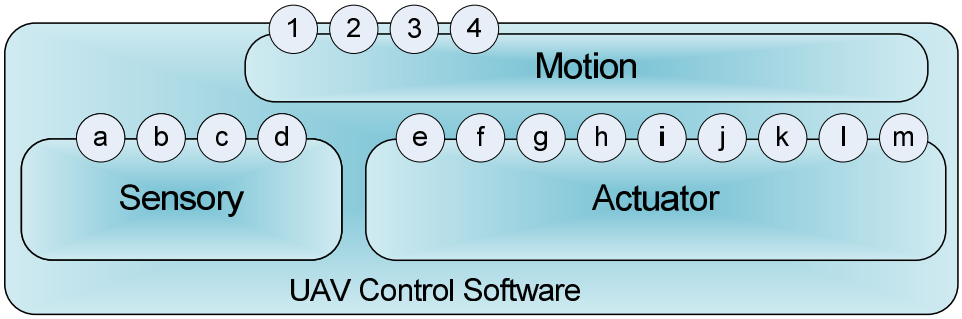


Figure 3.3: UAV Control Software Interface

provides direct access to sensors and actuators with a serial protocol through an RS-232 line. For example, to read the battery level of the robot, the command S \r where S is the name of the command and \r is the carriage return character is used. The robot returns the result as s, $battery\_level$ \n\r where s is the result type for the command S, $battery\_level$ is the battery charge level of the robot, \n is the line feed character and \r is the carriage return character.

The Koala robot can be extended with the use of the KoreBot [kt] module, which has a better processing power and provides a low-level robot control interface that can be used by high-level tasks. The operations a - m shown in Figure 3.3 are provided by lower level tasks, whereas the operations: (1) $moveForward(speed)$, (2) $turnLeft(speed)$, (3) $turnRight(speed)$, (4) $goTo(position)$ are provided by the higher level motion task, which uses the lower level tasks.

The low-level tasks provide operations for retrieving sensor readings and controlling the speed and position of the robot in a fine grained manner. For example, the $koa\_readProximity\ (device\_id, infrared\_sensor[\ ])$ operation provides the readings of 8 or 16, depending on the robot type, infra red proximity sensors while the operation

$koa\_setSpeed\ (device\_id, left, right)$ sets the left and right motor speed. A complete listing of operations a - m is shown in Appendix D.

In the previous sections, we have seen an overview of the management architecture. The rest of this chapter presents our mission specification approach. To illustrate the approach, we consider an example reconnaissance mission for determining whether an area is safe to be entered by humans. The following main roles are identified by the mission administrator: *Commander (C)*: controls the mission and allocates UAVs to roles. *Surveyor (S)*: explores the area and builds a map in terms of an occupancy matrix that indicates locations and the types of objects occupying that location – empty (no object), obstacle or hazardous material (chemical or biological). *Hazard-detector (H)*: detects hazardous chemical and biological substances. *Aggregator (A)*: aggregates information from all UAVs e.g., to produce a map showing the detected hazardous materials. Figure 3.4 shows a simplified view of the self-management framework applied to the example mission where a reconnaissance team comprising the *Commander*, *Surveyor*, *Aggregator* and *Hazard-detector* roles is formed. The mission specification part that relates to the mission layer is discussed in this chapter. The capability description and team formation parts that relate to the team layer will be discussed in the next chapter.

3.4 Roles

The role concept developed in this thesis is based on organisational roles. Organisations with a clearly defined mission (objective), rules and management structure are made up of positions known as roles that can be held by persons that satisfy the requirements of the positions [Slo94]. For example, a hospital may have an administrator, doctor, nurse and patient roles, which are populated with different persons at different times. The distinction between roles and their enactors is fundamental to adaptive systems management and a lack of this, as identified in [VSDF05], is a serious drawback in existing multi-agent systems.

The use of roles enables the specification of duties, rights and rules of interaction of persons enacting the roles irrespective of whoever is assigned to the role. It simplifies the specification of the organisation, allows changing persons (enactors of the roles)

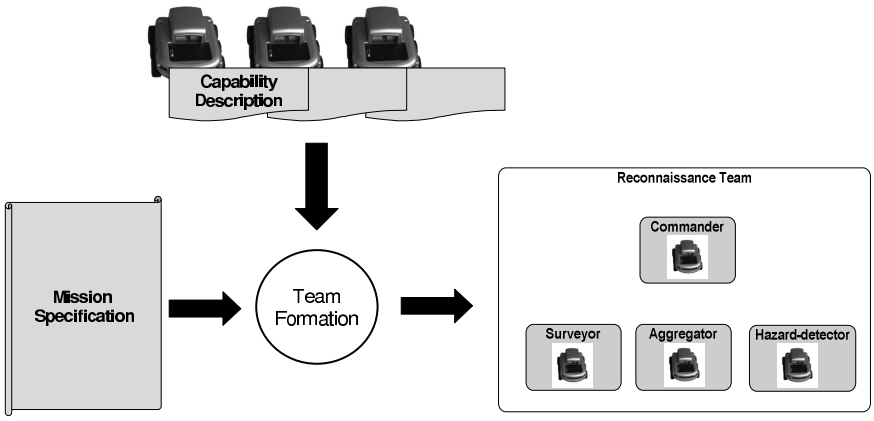


Figure 3.4: Overview of the Self-management Framework Applied for a Reconnaissance Mission

without changing the specification and enables specifying the organisation before recruiting persons to fill positions. We adopt this role-based approach for organising UAVs and creating dynamic teams. In our approach, a role is a placeholder to which discovered UAVs are assigned and is conceptually similar to an organisational role. However, by modelling the role in a manner that separates its local (individual) responsibilities and external (team-wide) responsibilities we present a novel approach that (1) enables fine grained control of individual as well as team-wide behaviours, (2) facilitates dynamic assignment of roles based on capabilities, and (3) enables a wide range of adaptations to context change and failure.

3.4.1 A Conceptual Model of Role

We model a role as a tuple comprising an external interface (E), a local interface (L), a role mission (R_M), a set of authorisation policies (A) and a set of tasks (T). E_P and L_P are the provided external and local interfaces respectively while E_R and L_R are the required external and local interfaces respectively. External interfaces relate to

remote interactions and local interface to internal interactions within the UAV.

$$R =< E,\ L,\ R_M,\ A,\ T > \quad \text{where}$$

$$E = E_P \cup E_R$$

$$L = L_P \cup L_R$$

These entities are able to specify what the role is expected to do, i.e., its functional behaviour as well as non-functional behaviours such as security, reliability and performance. Figure 3.5 shows a diagrammatic representation of a role.

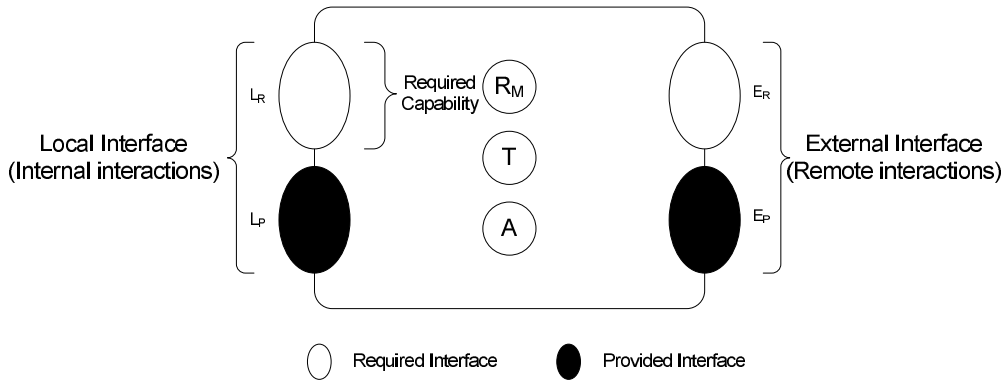


Figure 3.5: Role

The external and local interfaces provide a context for which role-mission (described later in this section) policies can be specified. Incoming events from the local or external interface can be used to trigger policies in role missions that invoke operations provided by the local and external interfaces. When a UAV is assigned to a role, the role-mission, authorisations and tasks associated with the role are loaded onto the UAV, unless already present.

Tasks of a Role

Tasks are complex operations that a UAV can perform – e.g., move from A to B, follow a path, track an object using video. The tasks in a role are usually inherent to the type of the role and hence are contained inside the role.

$$T = \{T_1,\ T_2,\ \ldots,\ T_k\}$$

Unlike policies, roles and missions, the specification of tasks is not done by the mis-

sion administrator as common tasks are provided by the autonomous system control software (tasks provided by the robot control software such as motion) and specific tasks for each domain of application (such as exploration) are designed by the domain expert. In essence, tasks in the role definition are place holders that are bound to available tasks at the mission execution time. The administrator needs to know only the interfaces of tasks in order to make the selection, configuration, specification of adaptation and exposition of the necessary events and operations to the role's external interface. We, however, make two assumptions: (1) tasks are modular (component based) to allow for different configurations and (2) a task either has an explicitly defined interface indicating its provided operations and notifications or can be queried for a description of its interface. These assumptions require legacy robot control software that may not meet these requirements to be wrapped by new tasks that transform them. Another point worth noting is that the role model considers only the higher level tasks (such as exploration and mapping), whose interfaces can be invoked by policies, and not the lower level tasks on which these higher level tasks depend (such as motion and sensory). This design choice is made in order to define the roles over a set of tasks at a high level of abstraction, which hides the heterogeneous nature of the UAV control actions specific to different types of UAV (e.g., operations specific to the Koala robot discussed in Section 3.3). There is, however, an overhead introduced by this choice. Since the roles are defined over an abstract set of interfaces instead of the exact UAV interfaces, an additional process, i.e., checking that the dependency requirements of the higher level tasks is satisfied by the UAV before the role is activated.

Obligation policies in the role mission may invoke operations supported by a task, activate or deactivate a task, change the behaviour of a task by changing its attributes, as well as reconfigure tasks. The *Surveyor* role in the reconnaissance mission, for example, consists of exploration (*Explore*) and map building (*BuildMap*) tasks (higher level domain specific tasks) as shown in Figure 3.6.

The exploration task has one provided interface (*ExploreI*) and three required interfaces, i.e., *CameraI, MotionI* and *BuildMapI*. Note that, in the figure, we have shown some elements of the provided interface, in detail, while showing only the interface types in the case of the required ones. This is because we will be using the provided interface elements to discuss the local and external interfaces of the *Surveyor* role.

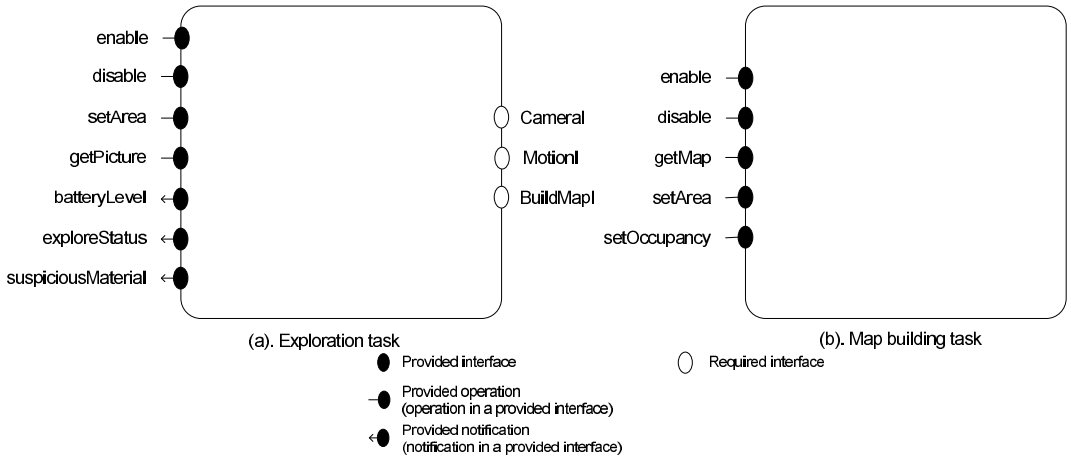


Figure 3.6: Tasks of a Surveyor role

The *Explore* task provides operations that allow for enabling (*enable( )*) and disabling the task (*disable ( )*), setting the area to be explored ($setArea(input = dmrc.util.Area)$) and taking a picture ($getPicture(output = dmrc.util.DmrcImage)$) where *dmrc.util.Area* and *dmrc.util.DmrcImage* are types for the input and output parameters respectively. It also provides notifications indicating battery status ($batteryLevel(name, level)$), coverage status ($exploreStatus(name, status)$) and the detection of a suspicious material ($suspiciousMaterial(name, location)$) where *name, status* and *location* are attributes showing the name of the task, the status (level) of the battery power, the status (coverage of the area) of the exploration and location of the suspicious material respectively. In addition, tasks have common management operations (not shown in the figure) for creating their own instance, and specific operations for binding their required interfaces to provided ones. For example, the *Explore* task has $create$, $bindCameraTask$, $bindMotionTask$ and $bindBuildMapTask$ operations.

Local Interface of a Role

The local interface defines the operations and events provided (L_P) by tasks within the local UAV and used by the role-missions, and the capability requirements of the role (L_R). It comprises:

- Operations provided by the tasks in the role – management operations such as activation and deactivation that are common to all tasks and task-specific operations.

- Events generated from within the role – management events generated by the role itself and operational events generated by the tasks within the role or propagated from the UAV components such as sensors and published via an event bus for use by other roles.
- Capabilities requirement, which is inferred from the tasks the role consists of, in the form of a summary (key words) or full description.

The operations in the local interface may be invoked by the policies in the role mission and the notifications may be used to trigger policies in the role mission or mapped to the external interface. The capabilities requirement is used by role assignment policies (discussed in Chapter 4) to check against the capabilities of a UAV. For example, the $setArea$, $getPicture$, $setOccupancy$ ($input = dmrc.util.Location$, $input = dmrc.util.HazardType$) and $getMap$ operations, and the $batteryLevel$, $exploreStatus$ and $suspiciousMaterial$ notifications from the tasks in the *Surveyor* role make up part of the provided local interface of this role. In addition, the provided local interface includes common management operations provided by the role such as $create$ and $assign$, specific operations such as $bindExploreTask$ and $bindBuildMapTask$, role assignment notifications which are generated when the role starts up ($newSurveyor$), i.e immediately after the role is assigned, and communication ($COMFailure$) and UAV failure ($UAVFailure$) notifications which are generated when communication links or UAVs fail if the role is managing other roles.

In addition to operations and events, the interface specification includes a capability requirement in terms of a list of services that must be provided by the tasks in the role. For example, from two of the tasks of the *Surveyor* role, we observe that the role requires a UAV that has motion and camera services. Assume the existence of a common motion interface that uses two sensory tasks, i.e., long-range and short-range distance sensors to detect the presence of far away and nearby objects, and assume that the camera task, which is a common interface as well, does not require any other tasks. The capability requirements of the role can then be summarised as *motion, camera, long-range distance sensor, short-range distance sensor*. This capabilities requirement makes up the required local interface of the role. A complete description of the requirement is generated from the required interfaces of the tasks included in the role. The capability summary is used in the first step of the vet-

ting process for role assignments and the full description is used to match the role requirements against the UAV provisions (discussed in Chapter 4). The operations, notifications and capabilities requirement discussed above define the local interface of the *Surveyor* role.

External Interface of a Role

The external interface defines operations and events relating to interactions with external collaborating roles. It consists of operations and events provided (E_P) and required (E_R) by the role to and from collaborating roles respectively:

- Management operations for loading missions, policies and tasks that are common to all roles.
- Operations from the local interface that are made visible to and can be invoked by other roles, i.e., by remote roles on other UAVs.
- Notifications from the local interface that are made visible and are disseminated to other roles.
- Operations that are required by this role. These operations are expected to be provided by collaborating roles.
- Notifications required by this role, generated by collaborating roles, e.g., to trigger policies.

A role may selectively expose some operations and notifications in order to allow other roles to use its service. Consider the *Surveyor* role. The $getPicture$ operation and $batteryLevel$ notification of this role may be exposed to other roles while the $setArea$ operation and $suspiciousMaterial$ notification are kept accessible only to the local policies. It may also expose the $UAVFailure$ notification generated as a result of its management functions. These operations and notifications make up the provided external interface of the role. The decision of whether to make the operations and notifications visible or not is made by the mission administrator during role specification.

In addition to providing notifications generated from management functions, a role may also require similar notifications such as $UAVFailure$ from other roles. For

example, the *Surveyor* role may need a hazard detection service whenever a suspicious material is detected; for this reason, it requires a $detectHazard(input{=}dmrc.util.Location, output{=}dmrc.util.HazardType)$ operation. These operations and notifications make up the required external interface of the role.

Role Mission

A role-mission specifies the functional behaviour of a role in terms of a set of obligation policies (event-condition-action rules) that allow for controlling tasks and other policies.

$R_M = \{O_1, O_2, \ldots, O_i\}$

These obligation policies are used to specify the mission of the role in an adaptive manner. They are used to specify: (1) the creation of task instances and/or the binding of the task place holders in the role to the task instances in the UAVs, and the configuration of tasks, (2) the activation or deactivation of tasks and/or invocation of operations provided by the role, (3) the activation or deactivation of other obligation policies (policy substitution), and (4) the assignment or reassignment of roles to UAVs. In Section 3.6, each of the first three mechanisms of adaptation are discussed in a greater detail and illustrated using the *Surveyor*'s role mission. The fourth type of adaptation is discussed in the next chapter.

Authorisations of a Role

In the previous sections, we have seen the different elements of a role in our model, i.e., its external and local interface, tasks and role missions. We will now see the last element in this model, i.e., authorisation policies.

$A = \{A_1, A_2, \ldots, A_j\}$

Authorisation policies specify how roles are permitted to interact with each other in terms of the events that can be triggered or operations that can be invoked via the external interface. Roles make some part of their provided local interface visible to other roles in order to facilitate collaborative missions. However, this type of visibility has a drawback of being visible to all or none, and hence there is a need to selectively

```
1 newauthpol := root/factory/authpolicy.
2 root/policy at: "sAuth" put: (newauthpol
3     subject: root/role/hdetector
4     action: "getPicture"
5     target: root/role/surveyor
6     focus:"t").
7 root/policy/sAuth reqneg.
8 root/policy/sAuth active: true.
```

Figure 3.7: Surveyor Authorisation Policy

permit access to some elements of the visible (external provided) interface based on the role types and/or current context. In addition, not all entities in the mission may be trusted to access specific services. This requires a means to explicitly specify access to these services to those that are trusted. Authorisation policies are used to achieve these objectives. For example, the *Surveyor* role's authorisation policy shown in Figure 3.7 denies access to the $getPicture$ operation for the *Hazard-detector* but allows the other roles, i.e., *Commander* and *Aggregator*.

3.4.2 Role Specification

A self-managing team comprises UAVs enacting various roles. Consequently, the formation of an adaptable self-managing team requires a role specification approach that allows modification of the role's functional as well as non-functional behaviours. A role has statically decided elements (E: external interface, L: local interface, T: tasks) that are set only during the specification and dynamic elements (R_M: role missions, A: authorisation policies) that can be changed in order to adapt its behaviour. In the previous sections, we have presented the different elements of the role model. In this section, we will present our approach for role specification. A role is defined, by the mission administrator, by specifying its elements using an XML role specification. The XML role specification corresponds to elements in the conceptual model of the role. It starts by identifying the type of the role using the $role$ tag and consists of the main elements identified by the tags – $policy$, $tasks$, $expose$, $local$, $require$ and $capability$. All publicly accessible operations of a task can be invoked by local policies hence the role specification needs to explicitly specify only those that should be exposed to remote roles (consequently remote policies) in order to make the specification succinct. However, should there be a need to selectively include only the necessary operations of the tasks into the local interface of the role, the $local$ tag is

used to indicate those operations. In addition, management operations such as *start*, *stop* and *assign*, which are common to all roles, are not included in the specification.

Figure 3.8 shows the specification of the *Surveyor* role we have been using in many of the previous examples. In order to fit the specification in one page, not all of the operations and notifications of this role used in the sample policies are shown.

The elements of the role specification are as follows:

- Policies relating to the role, i.e., both role missions (set of obligation policies) and authorisations (set of authorisation policies) are included within the *policy* tag. The inclusion can be through a resource identifier indicating the policies (if a policy repository is used) or the policies themselves escaped in XML's *CDATA* tag (if a policy repository is not used) or a combination of both as shown in the example specification Figure 3.8.

- The tasks in the role (i.e., the domain specific high-level tasks such as exploration and map building in our example mission) are indicated with the *tasks* tag. Each task's operations and notifications that are exposed to the provided external interface of the role are identified by the *operations* and *notifications* tag respectively and are put in the *expose* tag. Operations that give outputs, have a *result* tag identifying the type of the result, and operations that need inputs have the *arguments* tag identifying the type of the inputs. Notifications also identify their attributes. They do not need to identify the types of the attributes because all attributes are represented as strings. This design choice introduces a limitation since all attribute information that needs to be passed through notifications must be representable as a string. However, it is a deliberate tradeoff in order to limit the communication overhead that could be introduced by serialization-based direct passing of objects as attributes.

- The operations and notifications that are provided by the role itself (note that those from the tasks are specified within the *tasks* tag) and are exposed to the provided external interface are specified by the *expose* tag.

- The operations and notifications that are required by the role (required external interface) are specified by the *require* tag.

- The capabilities requirement of the role (required local interface) is specified by the *capability* tag.

After the administrator specifies the role using the XML based specification, the policy-managed role class code and the associated class code for the external interface of the role are automatically generated by a tool that is developed as part of the self-management framework (discussed in Chapter 7). This is possible because roles have statically defined external and local interfaces decided during the role specification. The design choice to make the role's interfaces static introduces limitation on the dynamic behaviour of the role since the domain of operations on which policies can be specified is static. On the other hand, this choice has a benefit of guaranteeing that policies specified for a set of roles cannot become invalid for the duration of the mission specification, which could be difficult to achieve if the role's interfaces were allowed to change dynamically.

3.5 Missions

A mission is a set of sequential or concurrent tasks that must be performed in order to achieve a goal. A planning process may generate more than one strategy for achieving a goal or the context may change such that the strategy for achieving the goal has to adapt to the current situation. The implication of this is that a mission specification for UAVs should allow adaptation of missions.

A team of UAVs should be able to perform a mission with a minimum number of UAVs that have the required capabilities although the configuration may not be optimal. When additional UAVs become available, the team should expand to make use of the new resources, thereby ideally optimising the non-functional behaviour of the team. Should there be a failure or departure of UAVs from the enlarged team, the team should contract but continue the mission. We define a *minimal team configuration* as the fewest types and number of UAVs needed to accomplish a mission. A *reasonably-optimal team configuration* has the ideal type and number of UAVs. A mission starts execution when a team satisfying the minimal configuration can be formed. The team will expand when additional UAVs join until it achieves the reasonably-optimal configuration.

```
<xml>
  <role name='Surveyor'>
    <policy uri="http://192.168.0.1/policy/surveyor">
     <![CDATA[//rate low policy
      policy:=root/factory/ecapolicy create.
      policy event: /event/frequentSuspiciousMaterial;
      condition:[:rate| rate < 5];
      action:[
      root/policy/checkHazardRandom active:false.
      root/policy/checkHazardMany active:true.].
     ]]>
    </policy>
    <tasks>
      <task name='Explore'>
        <expose>
          <operations>
            <operation name='getPicture'>
              <result>
                <name>picture</name>
                <type>dmrc.util.DmrcImage</type>
              </result>
            </operation>
          </operations>
          <notifications>
            <notification name='batteryLevel'>
              <attribute name='name'/>
              <attribute name='level'/>
            </notification>
          </notifications>
        </expose>
      </task>
      <task name='BuildMap'>
        <expose>
          <operations>
            <operation name='setOccupancy'>
              <argument>
                <name>hazardLocation</name>
                <type>dmrc.util.Location</type>
              </argument>
              <argument>
                <name>hazardType</name>
                <type>dmrc.util.HazardType</type>
              </argument>
            </operation>
          </operations>
        </expose>
      </task>
    </tasks>
    <expose>
      <notifications>
        <notification name='UAVFailure'>
          <attribute name='name'/>
          <attribute name='uav'/>
        </notification>
      </notifications>
    </expose>
    <require>
      <operations>
        <operation name='detectHazard'>
          <argument>
            <name>location</name>
            <type>dmrc.util.Location</type>
          </argument>
        </operation>
      </operations>
      <notifications>
        <notification name='UAVFailure'>
          <attribute name='name'/>
          <attribute name='uav'/>
        </notification>
      </notifications>
    </require>
    <capability>
      <require>
        <type>motion</type>
        <type>camera</type>
        <type>sonar</type>
        <type>infrared</type>
      </require>
    </capability>
  </role>
</xml>
```

Figure 3.8: Role Specification

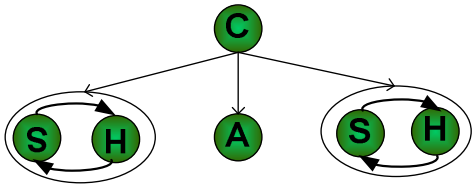


Figure 3.9: Reconnaissance Mission Minimal Configuration

This concept is illustrated using the reconnaissance scenario in which the minimal configuration is defined to be one *Commander*, one *Aggregator*, two *Hazard-detector*s, and two *Surveyor*s, where the *Surveyor* role is the primary role; and the *Hazard-detector* roles are secondary roles. Figure 3.9 shows this configuration. The lines with arrows emanating from the *Commander* role (C) indicate the direction of expansion of the team (assignment) and the lines with arrows connecting the *Surveyor* and *Hazard-detector* roles indicate that the UAV switches between these two roles.

These two roles are shown encompassed with a larger ellipse that represents the Role Manager discussed in Section 3.3. When a UAV is enacting a single role, the Role Manager is used only for bootstrapping a newly assigned role or during reassignment, and from then on the role takes full control of the UAV. However, when the UAV is enacting more than one role, the Role Manager will be active and facilitates the switching between roles based on policies. As shown in the figure (Figure 3.9), the *Surveyor* role is collocated with another role – *Hazard-detector*. The UAV has to switch between the *Surveyor* and *Hazard-detector* roles as only one of these can be active at a time.

A reasonably-optimal mission configuration is defined to be one *Commander*, two *Surveyor*s, two *Hazard-detector*s and one *Aggregator*. The team started with the configuration shown in Figure 3.9 and reached the configuration shown in Figure 3.10 as new UAVs join the team. The *Surveyor* roles, which assigned the *Hazard-detector* roles, serve as managers for those roles. Should any of the new UAVs fail or depart, the roles will revert to their minimal configuration position.

3.5.1 A Conceptual Model of Mission

A mission for a team of UAVs can be described in terms of roles, the relation among roles, constraints and mission parameters. The two basic relationships among roles

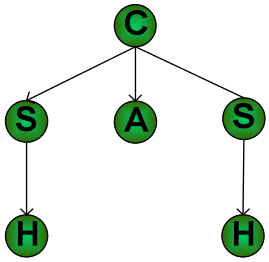


Figure 3.10: Reconnaissance Mission Reasonably-optimal Configuration

are *management* and *reporting*. We define the *manages* and *reports* relation between two roles x and y as $manages(x, y)$ where role x is responsible for assigning role y to a UAV and withdrawing it, and $reports(x, y)$ where role x is responsible for sending state information to role y periodically.

Mission parameters are used to specify global properties, which are shared by the roles in the mission. We consider two types of constraints, namely cardinality and collocation. The cardinality constraint sets the maximum number of roles a given type of role can manage and it is specified as a role-type and value pair. It has to be noted that the cardinality constraint also effectively specifies the maximum number of roles that can be assigned to a single UAV as part of a single mission. The collocation constraint indicates the type of roles that cannot be placed together. We consider negative collocation constraints (anti collocation) only because we do not have the notion of requiring two or more roles to be placed together since roles that have to be placed together can be designed as tasks and placed in a single role. Using the aforementioned entities, we define a mission, M, as follows:

$$M =< V, E, P, C > \quad \text{where}$$

$$V = \{R_1, R_2, R_3, \ldots, R_i\}$$

$$E = \{(x, y) | x \in V \wedge y \in V \wedge x \neq y \wedge manages(x, y) \wedge reports(y, x)\}$$

$$P = \{P_1, P_2, P_3, \ldots, P_j\}$$

$$C = \{C_1, C_2, C_3, \ldots, C_k\}$$

The set V consists of the roles in the mission. The set E is the set of management relationships among roles defined using the relations $manages(x, y)$ and $reports(x, y)$. The sets P and C contain the mission parameters and constraints respectively.

3.5.2 Mission Specification

In order to allow adaptation and reuse, we divide mission specification into three levels, namely policy, mission class and mission-class instance specifications. Policies are specified using Ponder2 [Pon] and stored in a policy repository. We have chosen Ponder2 as the policy specification language because of its support for both explicit specification of authorisation and obligation policies, which are necessary for our framework, its extensibility, and its scalability in that its interpreter is scalable enough to be deployed on systems with constrained resources. Our architecture, however, can easily be re-implemented to use any policy framework that supports explicitly defined event-condition-action rules and authorisation policies.

A mission class is an XML specification of constraints, mission parameters, types of roles needed for the mission and the management relation among the roles, while a mission class instance is an XML specification that defines the mission parameters and role cardinalities required to instantiate a mission class. The policy specification in the repository may apply to multiple mission classes, and there can be multiple instances of a mission instantiated with different parameters from a particular mission class.

Separately specifying policies, mission classes and mission class instances enables reuse. For example, consider the reconnaissance mission for which we will see example policies later in Section 3.6. A search and rescue mission can use a number of those policies (e.g., the adaptive cooperation pattern policies) as the policies defining the behaviour of the roles through task behaviour specifications, role interactions specifications, etc. can be applied to this mission. In addition, a mission class can be instantiated with different mission-class instance specifications thereby leading to reuse of mission class specifications. Figure 3.11 illustrates these points by indicating that both the reconnaissance and search & rescue missions can share policies specified and stored in the policy repository and that a reconnaissance mission class specification can be instantiated with different mission-class instance specifications based on the mission area.

The policy repository plays an important role by facilitating policy, mission class and other code reuse (tasks, roles). However, the repository is not central to the management framework as all specifications and code can be directly loaded to the

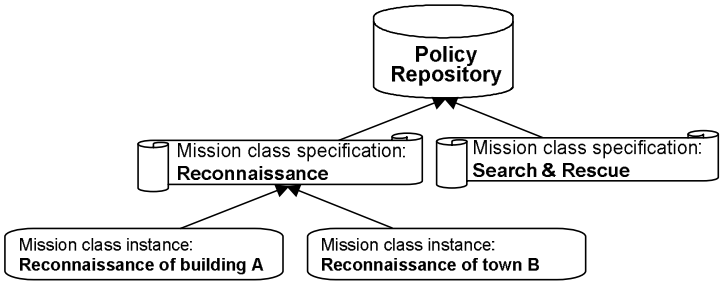


Figure 3.11: Mission Specification Levels

commander role from which it is distributed during role assignments thereby roles receiving all policies and necessary code related to the mission with the role assignment message. In addition, since the repository is comparatively small it can also be stored in the commander's memory (or any UAV's memory provided that it has the adequate resources) and policies may be distributed to other UAVs as needed or fetched by corresponding roles directly.

Policy Specification

The policies specified for a role are broadly divided into *role assignment policies*, used to assign UAVs to roles based on their capabilities and *operational management policies* used by roles to manage their own or collaborating roles' operational behaviour. The example policies we will later see in Section 3.6 are all operational management policies. In the next chapter, we will see role assignment policies.

Mission Class Specification

A mission class specifies a team in terms of roles, and policies that the roles use to manage themselves or other roles (where hierarchy exists) and indicates the management relation among the participating roles as well as the cardinality of each role. The cardinality and other parameters are instantiated later. Mission parameters such as failure-timeout that are shared by all roles are also included. This specification can be used to instantiate different teams of the same configuration with different cardinalities, mission parameters and role behaviours using policies. The policy-based role behaviour specification allows for changing the behaviours of assigned roles as illustrated in the example policies (Section 3.6).

```
<xml>
<missionClassParameters>
    <name>Reconnaissance</name>
</missionClassParameters>
<constraints>
    <cardinality/>
    <collocation/>
</constraints>
<missionParameters>
    <comTimeout>int</comTimeout>
    <failureTimeout>int</failureTimeout>
</missionParameters>
<commanderBehaviour>
    <roleManagement>
        <manages>surveyor</manages>
        <manages>aggregator</manages>
    </roleManagement>
</commanderBehaviour>
<surveyorBehaviour>
    <cardinality>int</cardinality>
    <roleManagement>
        <manages>hdetector</manages>
    </roleManagement>
</surveyorBehaviour>
<aggregatorBehaviour>
    <cardinality>int</cardinality>
    <roleManagement/>
</aggregatorBehaviour>
<hdetectorBehaviour>
    <cardinality>int</cardinality>
    <roleManagement/>
</hdetectorBehaviour>
</xml>
```

Figure 3.12: Mission Class Specification

Figure 3.12 applies to the reconnaissance scenario with a *Commander* role managing a *Surveyor* and an *Aggregator* role. The *Surveyor* role in-turn manages the *Hazard-detector* and *Relay* roles.

Mission Class Instantiation

A mission class instance (which gives rise to the actual team of UAVs performing the mission) specifies values for cardinalities, mission parameters and URIs of policies that define the role behaviour as shown in Figure 3.13. The mission administrator can specify policies relating to a mission in two ways – (1) through role specifications and (2) through mission instance specifications. The rationale for providing these two means lies in the nature of policies relating to a role. A policy relating to a role may involve only the role itself if it is triggered by a local event and invokes a local operation. On the other hand, an interaction policy relating to a role may involve external notifications and operations. Consequently, local policies can be specified

during the role specification while interaction policies can only be specified after the mission class is specified (the team structure and hence interactions are decided). However, specifying policies relating to a single entity at more than one point is prone to policy conflicts and hence we specify all the role's policies during the mission class specification.

```
<xml>
<missionParameters>
    <comTimeout>3000</comTimeout>
    <failureTimeout>7000</failureTimeout>
</missionParameters>
<commander>
    <cardinality>1</cardinality>
    <policy> http://192.168.0.1/policy/commander</policy>
</commander>
<aggregator>
    <cardinality>1</cardinality>
    <policy>http://192.168.0.1/policy/aggregator</policy>
</aggregator>
<surveyor>
    <cardinality>1</cardinality>
    <policy>http://192.168.0.1/policy/surveyor</policy>
</surveyor>
<hdetector>
    <cardinality>1</cardinality>
    <policy>http://192.168.0.1/policy/hdetector</policy>
</hdetector>
</xml>
```

Figure 3.13: Mission Class Instance Specification

3.6 Examples of Policy-based Adaptive Role Behaviour

Adaptive role behaviour through policy-based binding

The self-management architecture uses policies to perform late binding of the role's tasks to actual task instances. This approach enables us to instantiate roles that are more adaptive to the UAV's ability or context, to achieve a better performance. Although a UAV's capability is checked before a role is assigned to it, not all UAVs that satisfy the capabilities requirement have equal ability to perform different tasks. For example, consider a mission that requires two *Surveyor* roles and has two types of exploration tasks, namely *Explore* and *EnhancedExplore* (both implementing the same *ExploreI* interface discussed previously) where the latter provides a better result but requires more processing and battery power. Let us say that there are only two available UAVs and both have the capabilities *motion, camera, infrared, sonar* and hence satisfy the requirement for enacting the *Surveyor* role. Now assume one of

```
1  policy := root/factory/ecapolicy create.
2  policy event: /event/newSurveyor;
3  action: [:role :instance|
4  //Create tasks.
5  (root/task hasObject: "explore") ifFalse: [
6  root/task at: "explore" put:((root load: "dmrc.task.Explore") create).
7  ].
8  (root/task hasObject: "buildMap") ifFalse: [
9  root/task at: "buildmap" put:((root load: "dmrc.task.BuildMap") create)
10 ].
11 //Configure tasks.
12 root/task/explore bindBuildMapTask: (root/task/buildmap).
13 root/task/explore bindCameraTask:(root/task/camera).
14 //Bind tasks.
15 (root/role resolve: (role+"/"+instance))(
16     bindExploreTask: (root/task/explore).
17 (root/role resolve: (role+"/"+instance))
18     bindBuildMapTask: (root/task/buildmap).
19 ].
20 policy active: true.
```

Figure 3.14: Task Creation Policies

them has more processing power than the other. Including processing power in the capabilities requirement and vetting the UAVs accordingly is a feasible solution when there are a number of UAVs satisfying the capabilities requirement in order to achieve a better quality of service. Imposing a strict capabilities requirement that includes performance parameters may result in the mission not ever reaching its optimal configuration when there is a lack of UAVs that satisfy the requirement. In a scenario, such as our example, where there are not abundant UAVs to choose from, it is efficient to use policies to choose and load the exploration tasks based on the UAVs' processing and battery power after the UAVs are discovered. If we do so, both UAVs will be assigned to the *Surveyor* role but one will be loaded with the normal exploration task and the other (the powerful one) will be loaded with the enhanced exploration task resulting in the best possible outcome given the limited availability of UAVs. In the following, we will first illustrate how task bindings in the management framework are done using policies and then show an example policy that adaptively selects and loads an exploration task based on the UAV's battery and processing power.

We have previously seen that the *Surveyor* role contains the exploration and map building tasks. The policies shown in Figure 3.14 create instances of exploration (line 6) and map building tasks (line 9) using the classes *dmrc.task.BuildMap* and *dmrc.task.Explore*, which implement the *ExploreI* and *BuildMapI* interfaces respectively. The creation is performed only if the instances are not available (lines 5 & 8).

The required interfaces of the exploration tasks are then bound to the map building task (line 12) and camera task (line 13). Note that we have not tried to create the camera and motion tasks, which are not explicitly specified by the role. In addition, since these are low-level tasks which are necessary for the UAVs survival, they should already be active when the UAV is active. Now let us assume again that the camera task does not require any other tasks but the motion task does. This makes the exploration task completely configured except for the motion task configuration on which the exploration task depends. We will see later how the motion task is configured. For now, let us assume that the exploration task is completely configured. The *Surveyor* role instance is then finally bound to its two tasks, i.e., exploration and map building (lines 16 and 18). This policy is triggered by the $newSurveyor$ event (line 2), which has the attributes $role$ and $instance$. These attributes are used in the action part of the policy to programmatically determine the role type and instance (lines 15 - 18).

```
1  policy := root/factory/ecapolicy create.
2  policy event: /event/newSurveyor;
3  action: [:role :instance|
4  ((root/uav battery > 1000) & (root/uav processor > 400)) ifTrue: [
5  (root/task hasObject: "enhancedexplore") ifFalse: [
6  root/task at: "enhancedexplore"
7  put:((root load: "dmrc.task.EnhancedExplore") create)].
8  (root/role resolve: (role+"/"+instance))
9   bindExploreTask: (root/task/enhancedexplore).
10 ] ifFalse: [
11 (root/task hasObject: "explore") ifFalse: [
12 root/task at: "explore" put:((root load: "dmrc.task.Explore") create)
13 ].
14 (root/role resolve: (role+"/"+instance))
15  bindExploreTask: (root/task/explore).
16 ].
```

Figure 3.15: Adaptive Task Loading Policy

Now that we have the tasks loaded and created, let us go back to our example mission with two *Surveyor*s, and only two available UAVs. The policy shown in Figure 3.15 performs an adaptive task loading where, if the UAV has a battery power of more than 1000 milliampere hour and a processor with a speed greater than 400MHz (line 4), a better quality exploration task is loaded and the role is bound to this task (lines 7 & 9). Otherwise, the normal exploration task is loaded and the role is bound to this task (lines 12 & 15).

We recall that the exploration task has one required interface, i.e., the motion interface, that has yet to be bound. Assume the motion task requires two sensory

interfaces for short-range and long-range distance estimation. Infrared sensors perform better for short-range measurements while sonar sensors perform better for long-range measurements. Ideally, the *Surveyor* should be assigned to a UAV that has both sensors. However, in a situation where there is a shortage of capable UAVs, and instead mixed UAVs with some equipped with both sensors and some with either one of the sensors are available, the role performing the assignment (in our example the *Commander*) can relax the requirement (provided that it has policies that dictate so). This can be done without loosing the benefit of using the capable UAVs provided that the role being assigned (in our example the *Surveyor*) has policies that configure the motion task based on the context. The role being assigned, i.e., the *Surveyor*, does not need to have the knowledge of whether the assignment policy was relaxed or not. Figure 3.16 shows two policies that configure the motion task based on the available sensors. The first policy's condition is satisfied if both sensors are available (lines 7 & 8), and the motion task's required interfaces, namely *LongRangeDistanceSensor* and *ShortRangeDistanceSensor* are bound to the sonar (line 10) and infra red (line 11) tasks respectively. If it is the case that either one of the sensors is unavailable, the second policy's condition is satisfied (lines 17 & 18). Consequently, both the long-range and short-range distance estimations of the motion task will be supplied by either the sonar (lines 21 & 22) or the infra red (lines 24 & 25) based on the available type of sensor (line 20). In this section, we have seen that by using policy-based binding the management architecture achieves adaptive role behaviour. This approach allows for a wide range of adaptation including adaptation to component failures. For example, an event generated when an infra red sensor fails may trigger a policy to reconfigure the motion task to use the sonar sensor for its short-range distance estimation thereby allowing graceful degradation of services in the face of component failures.

Adaptive role behaviour through policy-based operation invocation

Policy-based invocation of operations is the primary mechanism for specifying cooperative missions in the self-management framework by means of role-mission policies that invoke operations on the external provided interfaces of collaborating roles. Consider the *Surveyor* role. We recall that this role has the $suspiciousMaterial(name, location)$ notification in its provided local interface and the $detectHazard\ (input =$

```
1  //Policy for configuring the motion task
2  //when both sensors are available.
3  policy := root/factory/ecapolicy create.
4  policy event: /event/newSurveyor;
5  //The capability entity, which is part of the management
6  //architecture, is queried for the UAV's capabilities.
7  condition: [((root/capability haslowcap: "sonar") &
8              (root/capability haslowcap: "infrared"))];
9  action:[
10 root/task/motion bindLongRangeDistSensorTask: /root/task/sonar.
11 root/task/motion bindShortRangeDistSensorTask: /root/task/infrared.
12 ].
13 //Policy for configuring the motion task
14 //when one of the sensors is not available.
15 policy := root/factory/ecapolicy create.
16 policy event: /event/newSurveyor;
17 condition: [((root/capability haslowcap: "sonar") &
18              (root/capability haslowcap:"infrared")) not];
19 action:[
20 (root/capability haslowcap:"sonar") ifTrue: [
21 root/task/motion bindLongRangeDistSensorTask: /root/task/sonar.
22 root/task/motion bindShortRangeDistSensorTask: /root/task/sonar.]
23 ifFalse:[
24 root/task/motion bindLongRangeDistSensorTask: /root/task/infrared.
25 root/task/motion bindShortRangeDistSensorTask: /root/task/infrared.].
26 ].
```

Figure 3.16: Adaptive Task Configuration Policies

$dmrc.util.Location$, $output$=$dmrc.util.HazardType$) operation in its required external interface. The role mission can then have a policy, shown in Figure 3.17, that is triggered by the $suspiciousMaterial(name, location)$ event, whose actions are (1) the invocation of the $detectHazard(input$=$dmrc.util.Location$, $output$=$dmrc.util.HazardType)$ operation, which is provided by another role, i.e., the *Hazard-detector* and (2) the invocation of the $setOccupancy(input$=$dmrc.util.Location$, $input$=$dmrc.util.HazardType)$ operation from the local interface of the role, using the location attribute from the notification and the result from the hazard detection process, as inputs. In essence, this policy specifies a cooperative action, in the example reconnaissance mission, triggered by a notification from the local interface of the *Surveyor* role and performed by both the role and its collaborator. In order for the *Surveyor* role's policies to be able to invoke operations provided by the *Hazard-detector* role, the *Surveyor* needs to have a remote role reference to the *Hazard-detector*. Policies should be specified to import role references depending on role interactions (examples shown in Chapter 6).

Figure 3.18 shows the interaction specified by the policy. The broken lines indicate the management relation between the two roles (discussed in the next chapter). The power of this approach and its benefits become more apparent when there are more

```
policy:=root/factory/ecapolicy create.
policy event: /event/suspiciousMaterial;
//If there is no entry called 'hdetector' then
//the condition is set to false as there is
//no hazard-detector
condition:[(root/role hasObject: "hdetector") ifFalse: [
false] ifTrue: [((root/role/hdetector size) == 1)]];
action:[:location :instance :role|
//if the number of hazard-detectors is 1
//get the hazard-detector role's
//reference (external interface).
hdetector := (root/role/hdetector listObjects) at:0.
//Invoke the 'detectHazard' operation on the hazardous
//material detector role.
hazardType := hdetector detectHazard: location.
//Update the map through the local interface
//of the surveyor role.
(root/role resolve: (role+"/"+instance))
    setOccupancy: location :hazardType.
].
//Name this policy for later reference.
root/policy at: "checkHazardSingle" put: policy.
policy active: true.
```

Figure 3.17: Cooperative Action Policy

collaborating roles.

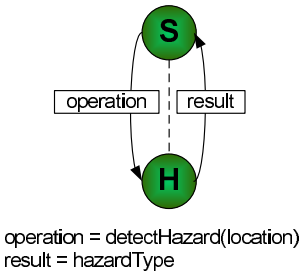


Figure 3.18: Interaction between the Surveyor and Single Hazard-detector role

Consider a scenario where there are more than one *Hazard-detector* roles. The *Surveyor* role now has a number of choices and can perform an adaptive choice of actions based on current context by using different cooperation patterns. It may (1) use a market-based pattern by announcing its location, receiving an estimate of distance from itself to each *Hazard-detector* role and then choosing the nearest *Hazard-detector* role, and making the invocation on the chosen *Hazard-detector* role (or use another measure such as sensor type, battery power, etc. instead of distance), (2) use a voting-based cooperation pattern by performing the invocation on some or all of the *Hazard-detector* roles and then choosing the result with the highest frequency (the mode) among the set of results (or use another statistical function, such as the mean, if applicable, depending on the type of the results), (3) use a randomly selected *Hazard-detector* among the available ones, and (4) use a combination of any one of

```
policy:=root/factory/ecapolicy create.
policy event: /event/suspiciousMaterial;
//If the number of hazard-detectors is > 5
condition:[(root/role hasObject: "hdetector") ifFalse: [
false] ifTrue: [((root/role/hdetector size) > 5)]];
action:[:location :instance :role|
//Invoke the 'measure' operation on all hazard-detector
//roles. The result is an array containing all the replies(results).
replies := (root/role/hdetector collect: [
    :name :hdetector| hdetector measure: location]).
//Get a copy of the reference (external interface) of all the
//hazard-detector roles in an array.
hdetectors :=(root/role/hdetector collect: [:name :value| value]).
//while(the size of the result array > 1){
//compare the result at index 0 with the result at the last index
//remove the greater from the results list
//remove the hdetector role reference at the same index as the greater
//result
//}
//the remaining hdetector is the one with the smallest
//distance (nearest)
[((replies size) >1)] whileTrue: [
 ((replies at:0) < (replies at:((replies size)-1))) ifTrue: [
    replies remove: ((replies size)-1).
    hdetectors remove: ((hdetectors size)-1).]
 ifFalse: [replies remove:0.hdetectors remove:0.].].
//get the role reference
selectedHdetector:=hdetectors at:0.
//invoke the 'detectHazard' operation on the selected hazardous
//material detector role
hazardType := selectedHdetector detectHazard: location.
//update the map through the local interface
//of the surveyor role
(root/role resolve: (role+"/"+instance))
    setOccupancy: location :hazardType.
].
//name this policy for later reference
root/policy at: "checkHazardMany" put: policy.
policy active: true.
```

Figure 3.19: Market-based Cooperation Pattern Policy

the above or other patterns.

Encoding cooperative patterns in the architecture limits its applicability in a wide range of missions as the cooperation pattern may differ in different missions or in different contexts within a mission. For example, voting may be useful but the overhead may become high when the number of *Hazard-detector* roles is larger. Similarly, the bidding-before-action approach of a market-based policy may be useful but disadvantageous as it has twice the communication overhead, which may be an issue if bandwidth is limited. Hence, the provision for the selection of the cooperation pattern based on current context is necessary.

In the following, we illustrate how policies in the role mission can specify adaptive cooperation patterns. Assume the reconnaissance mission has multiple *Hazard-detector* roles, and in addition to the operations we have seen so far, the *Surveyor*

role has the operation $measure\ (input{=}dmrc.util.Location, output{=}float)$ in its external required interface, and the *Hazard-detector* role has this operation in its external provided interface. The policy shown in Figure 3.19 specifies a market-based cooperation pattern between the *Surveyor* and the *Hazard-detector* roles. The *Surveyor* announces its position (similar to a bid) to all *Hazard-detector*s in its domain structure (to all that it is aware of) by invoking the operation $measure\ (input = dmrc.util.Location, output = float)$ on its required external interface, i.e., the *Hazard-detector*s' provided external interfaces. They reply with an estimate of their distance from the *Surveyor*; the *Surveyor* then selects the nearest *Hazard-detector*. The rest of the actions are similar to the single *Hazard-detector* policy we have seen in the previous example (Figure 3.17). The comments in the market-based cooperation policy (Figure 3.19) describe what each line does. Figure 3.20 illustrates the interactions specified by the policy

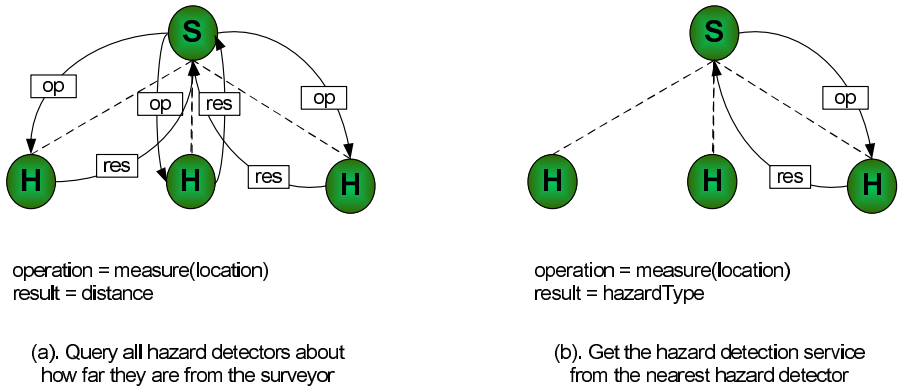


Figure 3.20: Interaction between the Surveyor and Multiple Hazard-detector roles

The *Surveyor* role can also use a voting-based cooperation pattern. In this case, it gets the hazard detection service from all the *Hazard-detector*s[2] and the result with the highest frequency of occurrence (the mode) is selected, and the occupancy map is updated accordingly, as shown in Figure 3.21.

The example policies for the single *Hazard-detector*, multiple *Hazard-detector*s with market-based cooperation and multiple *Hazard-detector*s with voting-based cooperation are defined on a disjoint set of conditions, i.e., *number of hazard-detectors = 1, number of hazard-detectors* > 1 *and* < 6*, number of hazard-detectors* > 5. Hence, at any time only one of these policies' conditions is met. In addition, these conditions

[2]This approach takes a much longer time since a synchronous method – $detectHazard(input = dmrc.util.Location, output{=}dmrc.util.HazardType)$ – which would take a longer time to return the result is used for hazard detection. These types of services are better implemented as event based ones in real-life applications. The case study chapter reconsiders the hazard detection service in that manner.

are checked only if there is a *Hazard-detector* role sub-domain (which is dynamically created when a UAV enacting that role joins the mission) in the *role* sub-domain of the domain structure. Hence, when there are no *Hazard-detector*s the policies do not execute their action section. Now consider a mission that has a variable number of *Hazard-detector*s throughout the mission execution due to failure, departure of UAVs, not enough UAVs arrived in the mission area yet, etc. The mission transparently adapts its cooperation pattern as the number of available *Hazard-detector* roles change since the cooperation patterns are selected by the three policies whose decision is based on the availability of these roles. Similar policies can be defined to achieve adaptation to different contexts.

```
policy:=root/factory/ecapolicy create.
policy event: /event/suspiciousMaterial;
//if the number of hazard-detectors is >1 and < 6
condition:[(root/role hasObject: "hdetector") ifFalse: [false] ifTrue:
[((root/role/hdetector size) > 1)&((root/role/hdetector size) < 6) ]];
action:[:location :instance :role|
//invoke the 'detectHazard' operation on all hazard-detector
//roles. The result is an array containing all the replies(results).
replies := (root/role/hdetector collect: [
    :name :hdetector| hdetector detectHazard: location]).
//update the map through the local interface
//of the surveyor role with the result that
//has the highest frequency of occurrence.
//the 'statistics' object is a helper object
//(implemented as part of the framework) that computes
//the mode and other statistical properties from a data set.
(root/role resolve: (role+"/"+instance))
    setOccupancy: (root/statistics mode: replies).
].
//name this policy for later reference
root/policy at: "checkHazardSeveral" put: policy.
policy active: true.
```

Figure 3.21: Voting-based Cooperation Pattern Policy

In the previous examples, we have seen how adaptive cooperative behaviour can be achieved through policy-based invocation of operations. Adaptive behaviour is also achieved through this mechanism by configuring task attributes that best suite the current context through the local provided interface of the role. For example, if the exploration task in the *Surveyor* role supports different search patterns, the role-mission policy of the *Surveyor* role can select different patterns based on the current context.

Adaptive role behaviour through policy substitution

Adaptation is further enhanced by using policies to substitute one or more policies as the context changes in multiple dimensions since the range of adaptation defined over the domain of one context (e.g., number of available *Hazard-detector*s) in a set of contexts may not be suitable to a change in another context (e.g., the frequency of occurrence of suspicious material in a given period of time). For example, our sample adaptive cooperation pattern policies cater for change in the number of available hazardous detectors. Now let us say the *Surveyor* role, in the interest of faster decision making, needs to use another pattern (shown in Figure 3.22), a random selection of one or more *hazard-detector*s, when it starts facing suspicious materials increasingly frequently.

```
policy:=root/factory/ecapolicy create.
policy event: /event/suspiciousMaterial;
//if there is no entry called 'hdetector' then
//the condition is set to false as there is
//no hazard-detector else
//if the number of hazard-detectors is >=1
//one is selected randomly
condition:[(root/role hasObject: "hdetector")];
action:[:location :instance :role|
//select the hazard-detector randomly
selectedHdetector:= root/statistics random:
    (root/role/hdetector listObjects).
//invoke the 'detectHazard' operation on the selected hazardous
//material detector role
hazardType := selectedHdetector detectHazard: location.
//update the map through the local interface
//of the surveyor role
(root/role resolve: (role+"/"+instance))
    setOccupancy: location :hazardType.
].
//name this policy for later reference
root/policy at: "checkHazardRandom" put: policy.
policy active: true.
```

Figure 3.22: Cooperative Action Policy (random selection)

In addition to the notifications we have seen so far, assume the *Surveyor* role has a $frequencySuspiciousMaterial(name, rate)$ notification in its provided local interface that is generated every specified period of time (e.g., 10 minutes) containing the rate of $suspiciousMaterial$ notifications within that period. Figure 3.23 shows policies that control other policies in order to adapt to current context. The first policy deactivates all the other cooperation patterns when the rate of suspicious material occurrence exceeds 10. The second policy reverts the system to the previous patterns if the rate goes below 5 (with a deliberate hysteresis to prevent the system from switching

patterns too frequently). Self-management is achieved by loading the system with a set of pre-specified policies, hence policy substitution is performed through activation and deactivation of policies that are already loaded in the system.

```
1  //rate high policy
2  policy:=root/factory/ecapolicy create.
3  policy event: /event/frequentSuspiciousMaterial;
4  condition:[:rate| rate > 10];
5  action:[
6  //deactivate policies
7  root/policy/checkHazardSingle active:false.
8  root/policy/checkHazardSeveral active:false.
9  root/policy/checkHazardMany active:false.
10 //activate policy
11 root/policy/checkHazardRandom active:true.
12 ].
13 policy active: true.
14 //rate low policy
15 policy:=root/factory/ecapolicy create.
16 policy event: /event/frequentSuspiciousMaterial;
17 condition:[:rate| rate < 5];
18 action:[
19 //deactivate policy
20 root/policy/checkHazardRandom active:false.
21 //activate policies
22 root/policy/checkHazardSingle active:true.
23 root/policy/checkHazardSeveral active:true.
24 root/policy/checkHazardMany active:true.
25 ].
26 policy active: true.
```

Figure 3.23: Adaptive Cooperation Pattern Policy

As illustrated in the examples, a wide range of adaptation is possible by specifying policies of the role mission that are available on the UAV from the point when the role starts (either received through the role assignment message, or fetched from the repository). However, provided that the UAVs have communication link to the policy repository, after the mission has started, an additional level of adaptation can be achieved through periodic checking of the policy repository performed by the roles for new policies. Also, an additional semi-automatic adaptation can be achieved if the mission administrator loads new policies to the UAVs directly. Although the latter approach requires manual external input (i.e., from the administrator), it, nevertheless, is useful since it still can make adaptation without shutting down the system.

3.7 Comparison with Related Work

As discussed in the background chapter, research on mission specification has resulted in a number of approaches. In this section, we compare our work to related

work in this area. For a more complete description of the related work the background chapter can be consulted. Also, note that we try to focus only on the mission management aspects of the related work with other aspects of the work, where applicable, considered in later chapters.

Mackenzie et al. specify missions for a group of robots in terms of behaviours (comparable to tasks in our framework) using CDL [MAC97]. Both individual and team behaviours are formed by statically composing behaviours. CDL does not have an explicit specification of team structure. However, it defines different cooperation schemes that implicitly define the team that is centrally coordinated. CDL also does not support adaptation of missions and lacks a means of specifying authorisations. It, however, allows for reuse of the mission specification. Our approach provides an adaptive mission specification and enables the creation of dynamic teams with an explicit specification of the team organisation (structure) that is necessary for capturing the management semantics in real-life missions. However, because CDL can compose both individual and team behaviours using a finite state machine, it can easily form a temporally sequenced team behaviour and is much more efficient for tightly coupled missions such as coordinated transport of objects.

CHARON [AGH+00] uses a similar behaviour-based composition approach to specifying missions and its specifications are reusable. It does not support adaptation of missions and explicit specification of team organisation, but does have support for access control.

ALLIANCE [Par98] specifies an adaptive mission for a group of robots using reusable pre-specified behaviour sets (composition of behaviours similar to CDL and CHARON) with motivational level thresholds, and mathematically modelled motivational behaviours (impatience and acquiescence) that activate or deactivate the behaviour sets when the motivational behaviour's output passes the motivational level threshold. ALLIANCE's adaptation is implicitly specified through the motivational level thresholds. Compared to our approach, the adaptation specification has a smaller configuration space since the threshold is a single parameter. Hence, the choice of a single threshold that captures the dynamics of the mission is necessary to get a meaningful adaptation. ALLIANCE addresses this issue using a learning mechanism to learn useful thresholds through the L-ALLICANCE [Par96] framework. Explicit specification of team structure and authorisations are also absent in ALLIANCE.

AYYLU [Wer00] models behaviours as a group of processes that have a common interface with ports that are externally accessible and specifies a mission in a reusable manner using a C-based language. It has an implicitly defined adaptation mechanism that is part of its task allocation scheme. The adaptation is based on broadcast of eligibility where each robot evaluates how eligible it is to perform a task and broadcasts this measure of fitness. AYYLU does not support explicit specification of team organisation and security.

TEAMCORE [TPC00] specifies a mission for a team of agents in terms of four sets of rules (using first order logic), namely joint intention, coherence constraints, intention tracking, and information-dependency relationship. The joint intention is comparable to our role-mission policies but it is more general as it can define both the team's state and commitment (goal) and also can be reasoned over to infer sub-missions from it. TEAMCORE, however, does not exploit this capability to form sub-missions and allocate them to sub-teams as it does not support hierarchical organisation of roles. Although the framework has the concept of team hierarchy, this is a goal decomposition where the leaf nodes are roles and in effect all roles are managed centrally. The coherent constraints, which enforce agents to follow a common solution path so that they do not hinder each other's effort to achieve the joint intention, are comparable to our interaction policies. The intention tracking rules, which enforce agents to monitor their peers, are comparable to management relationships in our approach. Our mission specification, however, does not have the equivalent of the explicit specification of the type of communication agents are allowed to perform, as done in TEAMCORE's information-dependency relationship (although this can be done with authorisation policies). TEAMCORE is powerful in its ability to specify a mission in a flexible and reusable way as well as allowing adaptation easily through the addition of new intentions to the team. However, although TEAMCORE's employment of the joint intentions method makes it robust, since as long as a goal is achievable, the team will achieve the goal eventually even in the face of context change and failure, it also makes it less reactive (slow) to context change and failure, on the other hand, since all team members have to reason and agree before they abandon the present solution path and enter a new one. This issue is especially crucial during failure in time sensitive applications and in applications where context change is frequent.

Tidhar [Tid93] proposed an approach known as social structures that deals with the specification of missions for a team of agents in terms of beliefs, goals and intentions. This approach shares the advantages of TEAMCORE as well as the limitations, i.e., the lack of immediate automated response to context changes and especially to failure. It, however, provides a more general framework of mission specification in multi-agent systems where the team can range from central to completely distributed organisation. Compared to our approach, it has, in a similar manner to TEAMCORE, a more general mission specification scheme that is based on first order logic but less reactive to context change as all the agents have to reason about the change, unlike the immediate adaptation achieved by policies that are triggered immediately after an event has occurred. The approach also does not take security into consideration.

OMNI [DVSD04, VSDF05] specifies missions for organisations in terms of social structures, scenes, social contracts and interaction contracts. OMNI's social structures are roles with objectives, rights and rules that are derived from the objectives and values of the organisation respectively. The role's obligations and rights are specified using norms and rules, which are described using deontic and dynamic logic respectively. Scenes, which are comparable to tasks in our framework, are processes performed by roles. Social contracts specify bindings of agents to roles and interaction contracts specify relationships between roles. Social contracts and interaction contracts are comparable to role assignments and management relationships respectively in our approach. OMNI's explicit representation of the team behaviour in terms of rules enables modifying the team behaviour without changing the agents. It differentiates between the roles and agents enacting roles thereby allowing norms to be specified without the knowledge of the agents enacting them. It also provides authorisations by specifying the rights of roles. In these aspects, it is similar to our approach. In a similar manner to TEAMCORE and Tidhar's social structures, OMNI has a more general approach to mission specification but unsuitable for dynamic teams such as those formed by mobile autonomous systems. However, OMNI's approach is even less responsive than the two. OMNI's use of deontic logic to represent rules, although allows it to elegantly describe obligations, makes the model computationally hard and hence less applicable in real-life dynamic missions.

3.8 Conclusion

In this chapter, we have presented an overview of the self-management architecture, which has three layers, namely mission, team and communication management and discussed the first layer, i.e., mission management. We have also presented the concepts used in the mission management layer and shown how they are used to specify and manage missions.

A novel role model that consists of tasks, authorisation policies and role-missions, which are sets of obligation policies, is presented. This model enables the specification of roles, which are used as building blocks of a mission, in terms of policies. Policies are used as the primary means of adaptation as they are used to modify role behaviours which in turn adapt the mission.

Missions for UAVs are specified in terms of roles with a separate specification used for the structure of the mission and its behaviour. This novel approach enables reuse of the mission specification with different behaviours resulting in teams of similar structure but different behaviour.

Chapter 4

Team Management

4.1 Team

More often than not, missions involve more than one UAV and hence the need for organising UAVs into teams in order to facilitate management and collaboration arises.

In our approach, missions are specified by a mission administrator in terms of roles, which are later populated by UAVs as discussed in Chapter 3. We define a team as this group of roles, enacted by UAVs. UAVs form a team in accordance with the mission specification, which defines a hierarchical structure among roles used for management purposes. Where in the hierarchy a specific role is placed in this structure is decided by the mission administrator (by means of the mission class specification) based on the type of the mission and the expected collaboration or patterns of interaction among different types of roles. Although the resulting team of UAVs has a hierarchical management structure, where a specific UAV is placed in the hierarchical structure is not pre-defined as it will depend on the order in which UAVs are discovered and their capabilities, which define the types of roles they can be assigned to.

Management interactions and messages (e.g., role assignments and state updates) make use of the hierarchical structure. However, collaboration interactions, i.e., operation invocations involving one role's policy and another role's external interface, are not constrained by the hierarchy since the role performing the invocation will

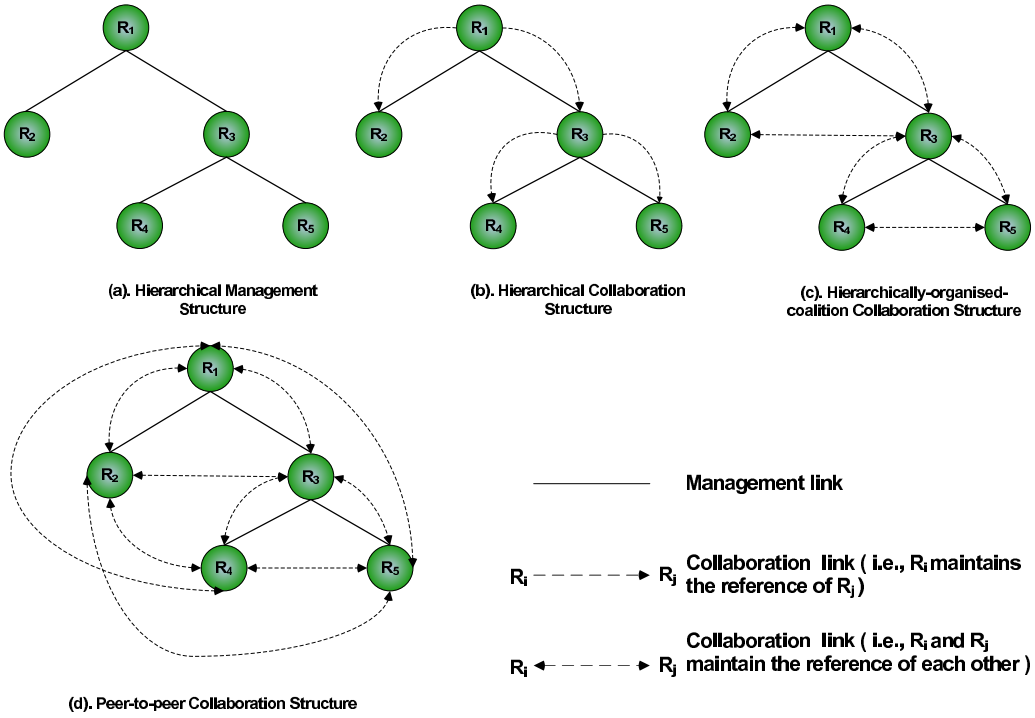


Figure 4.1: Organisation Structures

have a reference of the operation provider (the remote role) which it uses to access the operation directly. Since collaboration interactions are mission-dependent, the set of remote role references maintained by a role is determined by the role's policies. The hierarchical management structure facilitates the formation and maintenance of this interconnection among roles, which, in general, has an arbitrary organisational structure that could range from direct links formed only between each parent and its children roles in the hierarchy to a hierarchy of coalitions (a hierarchy of flatly organised group of UAVs) to a peer-to-peer structure, which directly links all roles, i.e., all roles maintain references to each other. Figure 4.1 shows an example of a hierarchical management structure with possible organisational structures for collaboration that could be formed and maintained by means of state update messages disseminated through the hierarchical management structure.

The rest of this chapter deals with the management structure. As mentioned before when to use which collaboration structure is mission-dependent and a discussion of the different possibilities is not within the scope of this thesis. However, how the mission-dependent interconnection of roles is specified through policies is shown in Chapter 6 and the mechanisms used to maintain this structure (referred to as domain structure) is presented in Chapter 7.

The type of organisational structure, i.e., the hierarchical structure, used for management is a characteristic of the management framework and hence it is common across all missions. Whether this structure should be centralised, peer-to-peer or hierarchical was a design issue. We chose the hierarchical structure for a number of reasons – (1) missions are inherently spatially distributed and a hierarchical organisation that would capture this distribution has the advantage of keeping local decisions and communications local thereby enabling faster decisions and causing less communication overhead, (2) it can distribute management responsibility thereby enabling scalability and robustness without the loss of the management semantics, (3) goals and hence missions are usually decomposed into trees, which makes the hierarchical structure a natural approach for role (task) allocation that can easily capture the goal decomposition tree, (4) it enables the creation of mission-dependent organisational structures used for collaboration (e.g., hierarchically organised coalitions, peer-to-peer structures, etc.) with minimal communication overhead by serving as a multicast tree. Using a hierarchical structure for management introduces a latency with respect to mission setup due to hierarchies. The hierarchical approach also renders the role at the root of the hierarchy a critical resource for the team. However, in our approach the role at the root of the tree (the commander) has a special importance only with respect to starting the team formation process. Once the commander has done so, it is only as important as the other roles since the role management is distributed.

4.1.1 A Conceptual Model of Team

A team of UAVs can be described in terms of the roles the UAVs are assigned to and the relationship among these roles, and it can be modelled as a digraph with the roles as vertices and their management relationship as edges. Using the $manages$ and $reports$ relationships defined in Chapter 3, we define a team, Γ, as follows:

$$\Gamma = (V, E) \quad \text{where}$$

$$V = \{R_1, R_2, R_3, \ \ldots, \ R_i\}$$

$$E = \{(x, y) | x \in V \wedge y \in V \wedge x \neq y \wedge manages(x, y) \wedge reports(y, x)\}$$

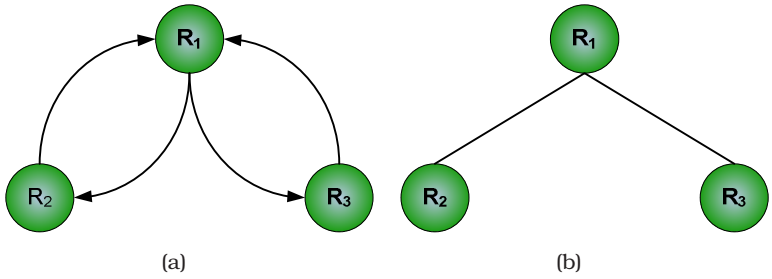


Figure 4.2: Team

The team model is related to the mission model defined in Chapter 3 in that V and E are also elements of the mission model. This is due to the fact that the mission specification also specifies the structure of the team. Figure 4.2(a) illustrates the digraph model of a team. We, however, will use a simplified version of this model, i.e., a simple graph that is also a tree, as shown in Figure 4.2(b) since it will serve the purpose for all practical cases we consider. In this figure, R_1 is responsible for assigning R_2 and R_3 which in turn are responsible for reporting their status to R_1.

4.2 Team Formation

The team layer of the management framework deals with the formation of a self-managing team of UAVs from a group of self-managing UAVs. It does so by discovering new UAVs within the communication range of those which are already part of the team, authenticating them [AGS+09], assigning them to suitable roles based on their capabilities, and maintaining the team. The team formation process is facilitated by five interacting services residing in the team layer. These are the *Capability Manager*, *Role Manager*, *Discovery*, *Optimiser* services and the *Role* itself that is being enacted by the UAV. The *Capability Manager* service is responsible for generating and advertising[1] the capability description of the UAV. The *Role Manager* deals with management tasks that are common for all roles such as loading, starting and stopping a role as well as providing support for running multiple roles on a single UAV. The *Discovery* service deals with discovery of new UAVs and the *Optimiser* service computes an optimal role assignment for the available roles and the discovered UAVs.

[1]The UAVs do not actually advertise their capabilities, they reply to discovery messages with their capabilities.

4.2.1 Discovery

UAVs discover each other using a discovery service that advertises the identification (ID) of a UAV periodically. Recall that we refer to a role (or a UAV enacting the role) that is responsible for assigning one or more roles as *manager role* and the roles managed by it as *managed roles*.

Discovery and assignment are done at every level of the hierarchy, i.e., by every manager role, as opposed to centrally by a single manager role at the root of the hierarchy.

A manager role runs a discovery service to discover new UAVs and assign them to one of its managed roles as illustrated in Figure 4.3. When a new UAV comes in to the communication range of the manager UAV, it will be able to receive the broadcast. The new UAV then replies to the broadcast with a join request if it is willing to take part in a mission. This leads to the initiation of a public-key based mutual authentication protocol that, if it succeeds, results in a shared secret key (described later in Section 4.2.2). Using this key, the discovered UAV will encrypt its capability summary and send it to the manager UAV. Encryption of capability information is necessary as, particularly in military applications, an adversary might use this information to infer mission goals and to plan attacks to neutralise a UAV's capabilities. Upon receiving the capability summary, the manager UAV, using its role assignment policies, decides whether the discovered UAV has the potential to be assigned to one of the roles and if so it will request the discovered UAV for its full capability description. If the manager UAV is satisfied with the full capability description, it will assign the discovered UAV to an appropriate role. The newly assigned role might run its own discovery service if it has roles to be assigned, and the discovered UAV will stop further capability advertisements since it is already enacting this role.

In the discovery protocol, a design issue considered was whether discovery messages should be limited to a single hop, i.e., only to nodes (UAVs) within the communication range of the discovering UAV, or be forwarded by intermediate nodes thereby allowing multi-hop discovery. We chose to limit the scope of discovery to a single hop for a number of reasons – (1) if discovery messages are forwarded, nodes that are not yet part of the team are to take part in the mission but since these nodes are not authenticated they cannot be trusted to participate in the mission, (2) ex-

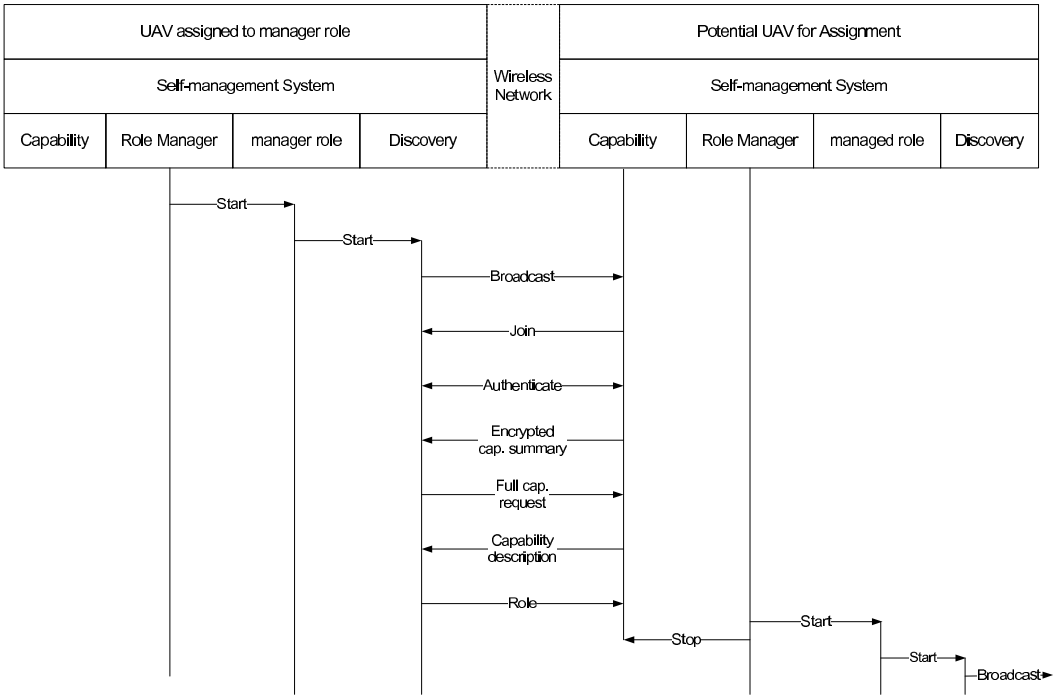


Figure 4.3: Discovery

isting team members may be enacting a manager role and hence will be trying to discover new nodes so forwarding other discovery messages would cause confusion and complicate the protocol, (3) forwarding discovery messages creates more network traffic, potentially causing congestion and depleting battery resources, and (4) missions often require members to be in the local geographical vicinity and requiring the nodes to be within radio range is a very simple way of enforcing this constraint. The single-hop based discovery, however, can lead to not finding a node with a particular resource that is not within the radio range but is available. It would be possible to cater for this at the application level by defining specific messages that are relayed to look for a scarce resource that is already part of the team and requesting it to move to where it is required.

Another issue was whether UAVs that are not yet team members should broadcast some form of announcement message rather than only respond to a discovery protocol. In our approach, the manager UAV, which is part of the team, instead of UAVs (nodes) that are not yet part of the team, broadcasts the discovery message in order to avoid unnecessary broadcasts by the non-member UAVs and enable immediate discovery of UAVs as they become available with a lesser impact on network traffic. A manager UAV has the knowledge of whether a UAV discovery is necessary or not as

it is aware of the mission roles and hence any discovery message broadcasted by a manager UAV is targeted to fill one of the roles by a UAV. In contrast, a non-member UAV has no knowledge of the mission, and hence the roles, before it is discovered and joined the mission. Consequently, if a non-member UAV was to broadcast discovery messages (announcements) in order to be discovered by a manager UAV, there will be instances where unnecessary broadcasts would be made by the non-member UAV, i.e., when all the mission roles that could be assigned by all the manager roles receiving the broadcast are already assigned. Broadcast by a manager UAV enables discovery of an available UAV as immediately as needed by the manager UAV. In contrast, broadcast by the non-member UAV makes the manager UAV wait for some time (the broadcast interval) before it learns about the available UAVs, unless broadcasts are too frequent (which is not a good solution as it overloads the network). In addition, announcement by non-member UAVs would waste energy and could be easily used as a form of attack, by malicious nodes, on manager nodes to get the managers to respond and waste energy.

4.2.2 Security

The UAVs in a team can change over time with new UAVs joining or leaving. These UAVs may also belong to different organisations (e.g., allies). Authenticating a UAV before it joins the team and protecting the ensuing communication is thus necessary to ensure the security of the mission, particularly for military applications. We assume the coalition between different organisations is achieved by using a Central Command Centre (C^3) that serves as a single certification authority (C^3), which issues certified public/private keys to all UAVs in the mission and maintains a Certificate Revocation List (CRL), and use the Certificate Public Key Infrastructure (C-PKI) [H+99] to ensure authentication, confidentiality and message integrity. The C-PKI system is used to exchange a common secret key generated using the Diffie-Hellman protocol [DH76] between each member of the team and its manager role. The secret key effectively establishes a secure channel between the manager role and its managed role. The steps involved in the authentication between a UAV (A) and the Commander (or any other manager role performing discovery) (C) are shown below.

1. C $\rightarrow$ A: $\{C_{id}\}$. Broadcast discovery message.

2. A $\rightarrow$ C: {Join Request, A_{id}, $Nonce_A$}. A sends a request to join the team.

3. C $\rightarrow$ A: Sign$\Big\{\{K_c\}_{K_{C^3}^{-1}}, C_{id}, Nonce_A + 1\Big\}_{K_c^{-1}}$. C authenticates itself to A by sending its Public-Key Certificate (PKC) and a function applied to $Nonce_A$, all signed with its private key.

4. A $\rightarrow$ C: Sign$\Big\{\{K_a\}_{K_{C^3}^{-1}}, A_{id}, Nonce_A + 2\Big\}_{K_a^{-1}}$. A sends its PKC to C as well as a function applied to the received Nonce, all signed by its private key.
   If the certificates are verified by both A and C (using C^3's certificate), mutual authentication is achieved.

5. C $\rightarrow$A: Sign$\big\{\{g^x \ mod \ p\}_{K_a}, g, p, Nonce_C\big\}_{K_c^{-1}}$. C sends the Diffie-Hellman parameters and keyshare encrypted with A's public key.

6. A $\rightarrow$ C: Sign$\big\{\{g^y \ mod \ p\}_{K_c}, Nonce_C + 1\big\}_{K_a^{-1}}$. A sends its Diffie-Hellman keyshare, encrypted with C's public key.
   Both A and C can now calculate a shared secret key (K_{ac}) that is used to establish a secure channel between A and C. The rest of the communication uses the secure channel established above.

There are some limitations in PKI system relating to the need for revocation lists. However, typical missions would last for hours to a few days at the most so certificates of participants in the mission will have been issued fairly recently before the start of the mission. Thus dealing with revocation was not considered an issue. Developing new security mechanisms was not a focus of this work.

4.2.3 Capability

The capabilities of a UAV indicate its ability and potential in terms of the basic resources and functionalities that are expected by the tasks and policies associated with the role that is to be assigned to the UAV. The assignment of a UAV to a role is based on the UAV's capability where the role definition specifies the capabilities required to support the role and only a UAV with the required capabilities is assigned to the role.

Capability Description

When devising a scheme for describing capabilities, the two major issues to be addressed are identifying the type of information to be provided by a capability description and specifying a representation language for the description. Based on the type of information they carry, we have identified two levels of UAV capability descriptions, namely *capability summary* and *full capability description*. A capability summary shows the low-level resources of the UAV and a summary of services provided by the UAV. For example, if we say that a UAV has a video camera, a long-range communication link, and a map builder service we are describing the UAV's hardware resources as well as services although the description does not give information on how to use them. The rationale for using this summary as a capability description is to provide information about the potential of the UAV, which influences the assignment decision (i.e., to which roles it can be assigned).

A full capability description defines the types of services a UAV provides and the list of associated operations and notifications relating to the use of the service. For example, a UAV which is capable of streaming a video might include in its service description that it can stream video, allows controls such as start, stop, tilt, pan camera, etc. UAVs provide different kinds of services such as mine detection, satellite communication, video streaming, etc. Roles assigned to these UAVs may need to use these services (tasks) as stand alone services or they may combine different services and create a new functionality. Whether or not a UAV can accommodate a role can be decided by checking its capability summary and full capability description.

Representing Capability Description

The representation of capabilities could be a simple text listing, a structured XML based description or even a set of first order logic formulas. Which representation mechanism to use depends on the type of information the description will carry and on the intended use of the description.

We found the UPnP [Upna] approach to service description to be the most fitting to our framework since it can describe both the capability summary and full capability. Our capability description has three main parts, a *system* section that provides general

information such as the name, owner, manufacturer, model, etc. of the UAV as well as credentials and contextual information such as battery power, a *device list* section that provides a list of the devices embedded in the UAV such as processors, sensors, etc., and a *service list* section, which describes the list of services provided by the UAV.

Unlike the UPnP service description, we specify the directions of all operations. This enables us to explicitly state what operations a service uses and what operations it provides thus enabling a dynamic service composition. In a basic service (non-composed service), the list of operations a service needs to access will be empty. In addition, we include an explicit statement of the notifications a service will accept and provide. Figure 4.4 shows an outline of a capability description.

Capability Matching

In parallel with our two levels of capability description, we have two levels of capability matching. The first one deals with matching the role's required capabilities (as specified in the local interface of the role) to the capability summary of the UAV in order to determine whether the UAV can accommodate the role or not. The result of this matching is: (i) Match: if the UAV has all the devices and services required by the role, or it has all the required capabilities and some more additional ones (superset match), (ii) No match: if the UAV does not have all of the devices and services required by the role. Note that a UAV that has only a subset of the required capabilities will be categorised as no match. For example, if one of the roles required capability is video streaming, and if the UAV has no camera then the UAV fails the first level of matching and consequently it will not be considered to enact that role.

There is a need to distinguish between exact match and a superset match when considering utility functions for optimising assignment of roles as discussed later.

The second one deals with matching the quality of service required by the role, and the tasks of the role with the full capability description of the UAV. The result of the quality of service matching may be used to adapt the behaviour of the role to the UAV's capability or it may lead to rejection of the UAV should there be a policy enforcing a minimum level of quality of service and the UAV's capability does not meet that. For example, if the UAV's full capability description is as shown in Figure

```
<xml>
<!-- General description -->
<system>
    <name>....</name>
    <manufacturer>....</manufacturer>
    <credential>....</credential>
    <battery>....</battery>
    <location>....</location>
    <!--other elements... -->
</system>
<!-- List of embedded devices -->
<deviceList>
    <device>
       <!-- video camera and its description... -->
    </device>
    <device>
       <!--hazardous chemical detector and its description... -->
    </device>
    <!--other embedded  devices.... -->
</deviceList>
<!-- List of services and their interfaces -->
<serviceList>
    <service>
        <operations>
            <in>
            <!-- operations it accesses-->
            </in>
            <out>
            <!-- operations it provides-->
                <name>operation1</name>
                <argumentList>
                    <argument>
                        <name>argument1</name>
                        <type>type1</type>
                        <allowedValueRange>
                            <minimum>minValue</minimum>
                            <maximum>maxValue</maximum>
                        </allowedValueRange>
                    </argument>
                    <!-- other arguments... -->
                </argumentList>
                <!-- other operations... -->
            </out>
        </operations>
        <notifications>
            <in>
            <!-- notifications it receives... -->
            </in>
            <out>
            <!-- notifications it provides... -->
            </out>
        </notifications>
    </service>
    <service>
       <!--operations and notifications of service2... -->
    </service>
    <!-- other services... -->
</serviceList>
</xml>
```

Figure 4.4: Outline of Full Capability Description

4.5 and the role requires a video streaming service, it may adapt the frame rate to the bandwidth of the UAV. On the other hand, if there is a policy that specifies that the resolution of the video stream should at least be 640×480 then this UAV will be

rejected. The task matching determines if the UAV has the necessary tasks relating to the role and if not it will provide a URI for the UAV so that it acquires the tasks before starting the role.

```
<xml>
  <system>
    <name>gunnersbury.doc.ic.ac.uk</name>
    <owner>Imperial</owner>
    <manufacturer>kteam</manufacturer>
    <battery unit='Ah'>3</battery>
  </system>
  <deviceList>
    <camera>
      <resolution unit='px'>
        <width>320</width>
        <height>240</height>
      </resolution>
    </camera>
    <wifi>
      <bandwidth unit='Mbs'>1</bandwidth>
    </wifi>
  </deviceList>
  <serviceList>
    <dmrc.task.Explore>
      <out>
        <notifications>
          <batteryLevel/>
          <exploreStatus/>
        </notifications>
        <operations>
          <disable>
            <arguments/>
          </disable>
          <enable>
            <arguments>
              <type>dmrc.util.Area</type>
            </arguments>
          </enable>
        </operations>
      </out>
    </dmrc.task.Explore>
    <dmrc.task.HazardDetect>
      <out>
        <notifications>
          <hazardStatus/>
        </notifications>
        <operations>
          <start>
            <arguments/>
          </start>
          <stop>
            <arguments/>
          </stop>
        </operations>
      </out>
    </dmrc.task.HazardDetect>
  </serviceList>
</xml>
```

Figure 4.5: Example Full Capability Description

Both the capability summary and full capability matching assume an agreed ontology between the discovering UAV and the discovered UAV.

4.2.4 Role Assignment

A UAV is assigned to one of the roles in the mission based on its capabilities, which it provides to the managing role during the discovery process. The managing role checks the capability and decides whether to assign it to one of the roles or not and then to which role to assign it to. The decision can be made either immediately or in a delayed manner giving rise to an immediate or optimised role assignment respectively.

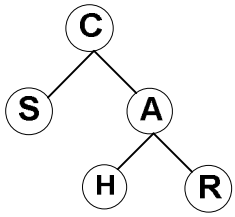


Figure 4.6: Reconnaissance Team

Management Tree

The UAVs in a mission are arranged in the form of a management tree during the role assignment process to facilitate decentralisation with any of the UAVs potentially performing discovery and role assignment. This tree is used for defining management hierarchies as well as data aggregation during execution of the mission. Figure 4.6 shows a management tree for the reconnaissance mission described in Section 3.2, consisting of the *Commander*, *Surveyor*, *Aggregator*, *Hazard-detector* roles and one additional role type, i.e., a *Relay* (R) role that facilitates communication. In the following sections, we present the management tree formation algorithms. Each UAV, upon startup, runs the *Managed UAV's algorithm*. However, if the UAV is started as a commander, it runs the *Manager UAV's algorithm*. In the event that a UAV becomes a manager, it also runs the *Manager UAV's algorithm*.

Manager UAV's Algorithm

A manager role has a set of roles it is required to assign according to the mission specification. When the role is started, it prepares a waiting list (W) containing a set

of roles to be assigned to UAVs : $W = \{R_1, R_2, ..., R_n\}$ and a children list (L) containing a set of assigned roles and their state information.

1. Broadcast ID periodically to discover other UAVs.
2. If a UAV replies with a join request, the manager initiates a mutual authentication process that, if successful, will result in a shared secret key between the managing and managed UAVs. If the authentication [AGS+09] is not successful, return to step 1.
3. The authenticated UAV sends an encrypted capability summary s, check if there is any role in W with a role assignment policy specifying a capability requirement r, where $r \subseteq s$. If there is such a role then send a request for a full capability description to the UAV.
4. Check if the full capability description satisfies the requirements of the role and if so send a role assignment message to the UAV. Remove the assigned role from W and add it to L.
5. If a state update message is received, update L.
6. Check L for freshness of role state[2] information. If the age of the state of a role is higher than a given interval of time then publish an appropriate failure event[3] (this could be communication link failure or UAV failure event based on the age of the state). Return to step 5.

For all UAVs that have responded to the broadcast, steps 2-4 of the above algorithm execute in parallel.

Managed UAV's Algorithm

1. Wait for broadcast.
2. If a broadcast message is received and if this UAV is willing to participate in the mission, send a join request to the broadcaster.

[2]The role state information may range from an empty message only used to let the manager role know that the role is alive to a message that contains a domain specific information such as the location of a hazard.

[3]The failure events are used by the failure management policies that subscribe to these events.

3. If authentication is initiated by the broadcaster, then perform mutual authentication. If the authentication is successful, send an encrypted capability summary to the broadcaster else return to step 1.

4. If a full capability request from the broadcaster is received within a given timeout then send the encrypted description else return to step 1.

5. If a role assignment message is received within a given timeout then download the policies specifying the behaviour of the role, from the manager or any other node acting as a policy repository; start the role and identify the broadcaster as the parent (manager) UAV else return to step 1.

6. Send a state update message to the manager UAV periodically.

Figure 4.7 illustrates a trace of the tree formation algorithm. Figure 4.7(a) shows neighbouring UAVs forming an ad hoc network. In Figure 4.7(b), the top UAV broadcasts discovery messages to its neighbours, which eventually form a team with the top UAV as commander and the middle UAVs as children assigned to various roles (Figure 4.7(c)). In Figure 4.7(d), the middle UAVs broadcast discovery messages to their neighbours but only lower UAVs respond as the other middle UAVs already have a parent. Figure 4.7(e) shows the resulting tree with each UAV having a single parent.

Immediate Role Assignments

An *immediate role assignment* implies a decision where a discovered UAV is immediately assigned to one of the available roles without considering other roles and possible future discoveries. Consider again the reconnaissance team shown in Figure 4.6 with roles *Commander* (C), *Surveyor* (S), *Aggregator* (A), *Relay* (R), *Hazard-detector* (H), with their capability requirements shown in Figure 4.8(a).

Assuming that the *Commander* role is already assigned at time t_0 this mission still needs four UAVs. Now let us assume that four UAVs, with capabilities as shown in Figure 4.8(b), come within the *Commander*'s communication range with the arrival order shown in Figure 4.9. UAV_1 is discovered at time t_1 and the *Commander*'s role assignment policies, shown in Figure 4.10, dictate that it be assigned to either a *Surveyor* or an *Aggregator* role. Although UAV_1 has more capabilities than required

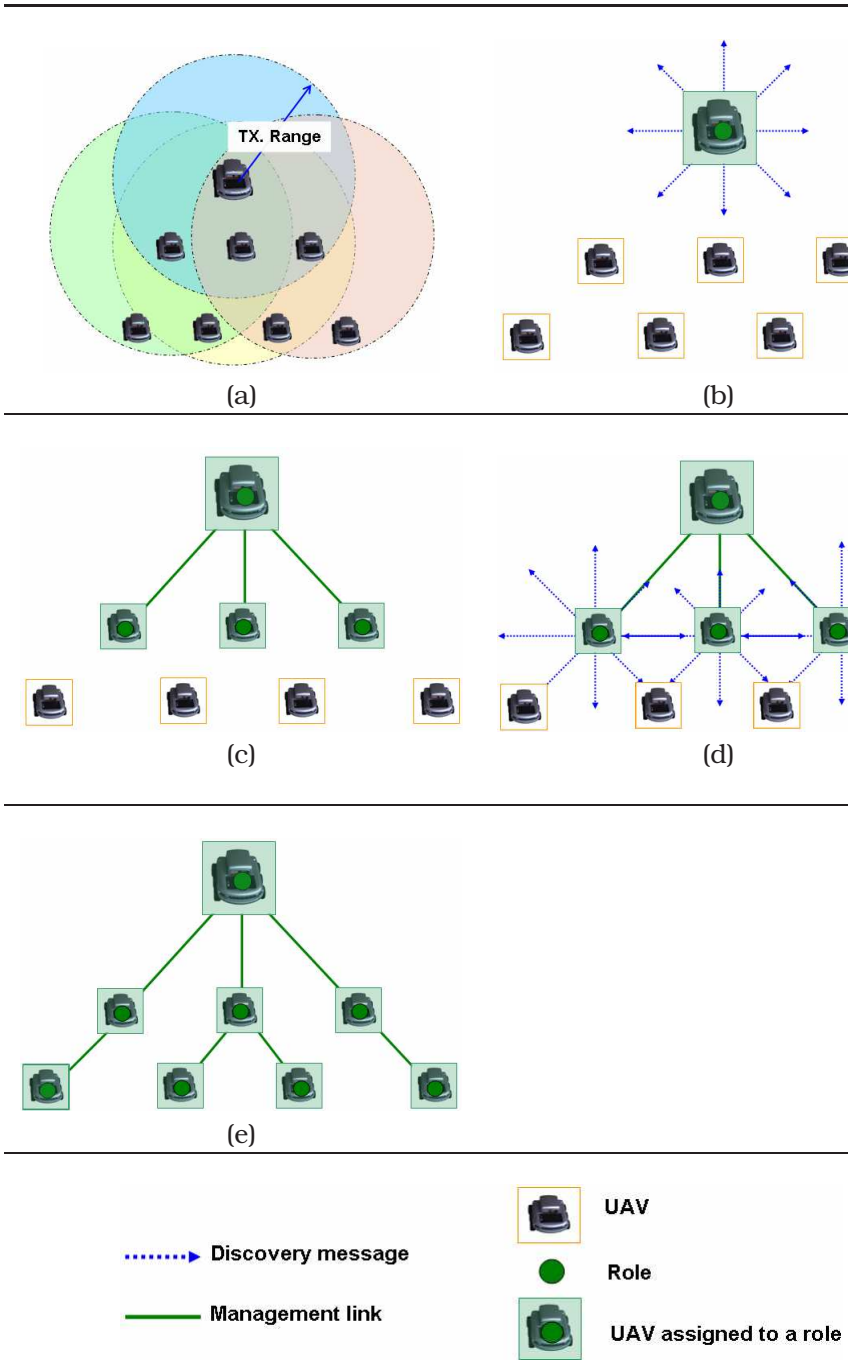


Figure 4.7: Management Tree Formation

by a *Surveyor* role, since it satisfies the capability requirement of both roles it can be assigned to either one of them. Assuming the worst case scenario, i.e., a UAV with scarce resource being assigned to a role that does not need the scarce resource, let us say that it is assigned to the *Surveyor* role. UAV_2 is discovered at time t_2 and it will be assigned to the remaining role, i.e., the *Aggregator*. UAV_3 and UAV_4 are discovered at times t_3 and t_4 respectively by the *Aggregator*, which is now running a

Role Type	Required
S	{video, motion}
A	{mapping}
R	{motion, longrangecom}
H	{motion, hdetection}

(a) Capability requirement of roles

UAV	Provided
UAV_1	{mapping, video, motion}
UAV_2	{mapping, video, motion, longrangecom}
UAV_3	{video, motion}
UAV_4	{motion, hdetection}

(b) UAV Capabilities

Figure 4.8: Capabilities

discovery service since it needs to assign two roles. However, UAV_3 does not satisfy the capability requirement of either one of them and hence it will not participate in this mission. UAV_4 will be assigned to the *Hazard-detector* role. The *Relay* role will never be assigned thereby leaving the mission in an incomplete team configuration until a UAV, if any, with the necessary capabilities is discovered.

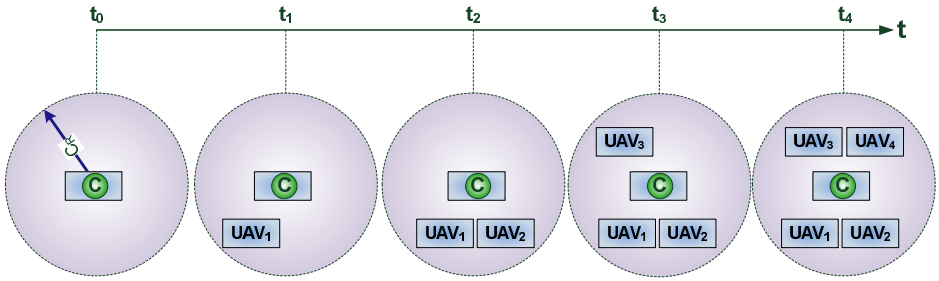


Figure 4.9: UAV Arrivals

Consider two UAV arrival orders different from the one shown in Figure 4.9, $UAV_1 \rightarrow UAV_3 \rightarrow UAV_2 \rightarrow UAV_4$ and $UAV_2 \rightarrow UAV_1 \rightarrow UAV_3 \rightarrow UAV_4$ where $a \rightarrow b$ stands for *UAV a is discovered before UAV b*. Using the role assignment policies of the *Comman-*

```
1 on discovered(UAV, credentials, capability)
2 do assign(UAV, "Surveyor")
3 when authenticated(credentials) and
4       capability.hasCapabilities("motion","video")
5
6 on discovered(UAV, credentials, capability)
7 do assign(UAV, "Aggregator")
8 when authenticated(credentials) and
9       capability.hasCapabilities("motion","mapping")
```

Figure 4.10: Commander Role Assignment Policies

```
1 on discovered(UAV, credentials,capability)
2 do assign(UAV,"Relay")
3 when authenticated(credentials) and
4      capability.hasCapabilities("motion","lrangecom")
5
6 on discovered(UAV,credentials,capability)
7 do assign(UAV,"hdetector")
8 when authenticated(credentials) and
9       capability.hasCapabilities("motion","hdetection")
```

Figure 4.11: Aggregator Role Assignment Policies

der and the *Aggregator*, shown in Figures 4.10 & 4.11, in the worst case scenario, the team may end up with assignments $\{S \mapsto UAV_1,\ A \mapsto UAV_2,\ H \mapsto UAV_4,\ R \mapsto \emptyset\}$ and $\{S \mapsto UAV_2,\ A \mapsto UAV_1,\ H \mapsto UAV_4,\ R \mapsto \emptyset\}$ respectively where $r \mapsto s$ stands for *role r is assigned to UAV s*. In both cases, the team ends up with an incomplete configuration.

In the first case, the problem was initially caused due to the fact that when deciding to assign the *Surveyor* role to UAV_1 the *Commander* did not take the other role (*Aggregator*) waiting to be assigned into consideration thereby leading to a later assignment of this role to a UAV that has a scarce capability[4] (i.e., long-range communication) required by the *Relay* role. This problem could have been avoided if the *Commander* had taken all its roles waiting to be assigned into consideration instead of immediately assigning discovered UAVs.

In the second case, the problem was caused by the fact that the *Commander* has again used a UAV with a scarce capability needed by the *Relay* role. However, this problem could not have been avoided even if the *Commander* had taken all its waiting roles into consideration since both UAV_2 and UAV_1 satisfy the capability requirement of the *Surveyor* and *Aggregator* roles.

From the scenarios discussed above, we can observe that immediate role assignment leads to incomplete team configurations because of the lack of local (the *Commander* not considering the *Aggregator*) and global (the *Commander* not considering the *Relay*) consideration of future role assignments.

[4]A capability that is not provided by most available UAVs.

Optimising Role Assignments

The team formation starts from the UAV that is initially loaded with the mission specification. This UAV assigns roles managed by it, and these roles in turn repeat this process until the team reaches its final configuration. The discovery as well as optimisation algorithms work in a recursive manner on every UAV, at any level of the hierarchy, that manages other UAVs. If the roles take local and global future assignments into consideration, the team will reach its optimal configuration provided that UAVs with the necessary capabilities are available. To attain this we define two objectives a manager role should aspire to achieve:

- Local objective: a manager role should work towards assigning all its managed roles to the best available UAVs.
- Global objective: a manager role should work towards facilitating the assignment of all roles managed by its managed roles.

We model the role assignment problem using roles, UAVs, the management relation among roles, constraints and utilities as shown in Figure 4.12. The functions req_{cap} and $prov_{cap}$ express the required and provided capabilities by roles and UAVs respectively. $battery(s)$ gives the available battery power of a UAV. Management relations are expressed through the *managed by* (represented as $mgdby$) and management closure (represented as $mgmnt_{closure}$) relations. $utility(r)$ gives a measure of the benefit acquired by using a role r. For example, in a mission where roles have different priorities the utility of a higher priority role is more than that of the one with a lower priority. $utility(s)$ gives a measure of the benefit acquired by using UAV s. For example, a UAV with more battery power has higher utility than the one with a lower battery power. $utility(r, s)$ gives a measure of the benefit acquired by assigning role r to UAV s. For example, if a manager role is concerned with global objectives the utility of an assignment of a role to a UAV that has the exact capability requirements is higher than that of the one using a UAV with more capabilities than required by the role. This will decrease the probability of reaching an incomplete team configuration by preventing the use of more capable UAVs for less demanding roles. On the other hand, if a role is concerned only with local objectives the utility may be the same for all assignments that satisfy the capability matching requirements. $weight(utility)$

1. A set R of n roles with their attributes

$$R = \{r_1, r_2, r_3, \ldots, r_n\}$$
$$A \quad set \quad C \quad of \quad t \quad capabilities.$$
$$C = \{c_1, c_2, c_3, \ldots, c_t\}$$
$$req_{cap} : R \rightarrow \mathcal{P}(C) \quad \text{where } \mathcal{P}(C) \text{ is the power set of } C$$

2. A set S of k UAVs with their attributes

$$S = \{s_1, s_2, s_3, \ldots, s_k\}$$
$$prov_{cap} : S \rightarrow \mathcal{P}(C)$$
$$battery(s) : S \rightarrow \Re^+ \text{ where } \Re^+ \text{ denotes the set of non-negative real numbers}$$
$$cred(s) = \text{credentials of a UAV}$$

3. A management relation among roles

$$mgdby : R \rightarrow \mathcal{P}(R) \quad \text{where}$$
$$r_i \in mgdby(r_j) \quad \text{if} \quad r_j \quad \text{is responsible for assigning} \quad r_i$$
$$management_{closure} : R \rightarrow \mathcal{P}(R)$$
$$management_{closure}(r_j) = \begin{cases} r_j & \text{if } mgdby(r_j) = \emptyset \\ \bigcup_{r_i \in mgdby(r_j)} management_{closure}(r_i) & \text{otherwise} \end{cases}$$

4. A set of constraints that must be satisfied by a valid role assignment.

$$colloc : R \times R \rightarrow \{-1, 0, 1\}$$
$$colloc(r_i, r_j) = \begin{cases} -1 & \text{if } r_i \text{ and } r_j \text{ cannot be assigned to the same UAV} \\ 0 & \text{if there is no restriction} \\ 1 & \text{if } r_i \text{ and } r_j \text{ have to be assigned to the same UAV} \end{cases}$$

5. A set of utility functions providing a measure of the benefit acquired by performing any assignment.

$$utility(r) : R \rightarrow \Re^+$$
$$utility(s) : S \rightarrow \Re^+$$
$$utility(r, s) : R \times S \rightarrow \Re^+$$
$$weight(utility(r)) : U_{role} \rightarrow \Re^+$$
$$weight(utility(s)) : U_{UAV} \rightarrow \Re^+$$
$$weight(utility(r, s)) : U_{role,UAV} \rightarrow \Re^+$$
$$U_{role} = \{utility_1(r), utility_2(r), utility_3(r) \ldots utility_m(r)\}$$
$$U_{UAV} = \{utility_1(s), utility_2(s), utility_3(s) \ldots utility_p(s)\}$$
$$U_{role,UAV} = \{utility_1(r, s), utility_2(r, s), utility_3(r, s) \ldots utility_q(r, s)\}$$

Figure 4.12: The Role Assignment Model

provides the weight associated to each utility function. For example, in one mission scenario a utility measuring a battery power may be given more weight than a utility that measures the degree of capability matching. However, this choice may change

in another scenario and hence the need to model utility weights as functions. The weights of the utility functions are decided by the mission administrator and specified, using policies, as part of the mission specification. However, this decision of what weights to use for each utility function, and defining the utility functions themselves is a difficult task. It is an iterative process in that the weights and functions need to be modified based on experience and needs a thorough knowledge of the domain of the mission.

The node mobility model is not considered for optimisation. Once the node joins the mission, the mobility pattern of the node will be controlled by the mission policies. Prior movement patterns are irrelevant. However, mobility attributes such as battery power, maximum speed, maximum incline that the vehicle can cope with, hovering capability for airborne vehicles, etc. are treated as capabilities when making role assignment decisions.

Using the role assignment model shown in Figure 4.12, we formulate the role assignment problem as shown in Figure 4.13 where u_i denotes $utility_i$ and w denotes $weight$. In this formulation, the aggregate utility of each discovered UAV with respect to available roles is computed. The utility is set to 0 if either the utility of using the UAV (system utility) or the utility of using the UAV for a certain role (role-system utility) is 0. This is because if the utility pertaining to a UAV is 0 in one aspect then the UAV should not be used (e.g., not match, battery level lower than the minimum allowed level, etc.). Once the aggregate utility for all discovered UAVs and available role is computed, the next step is solving the assignment problem. To solve this problem, we model it as a weighted bipartite-graph minimum-cost maximum-matching problem where one partition of vertices are the roles and the other partition of vertices are the UAVs. The weights of the edges in the graph are computed using the utility (treating maximum utility as minimum cost) for the role-UAV pair.

Efficient algorithms, such as Hopcroft and Karp's $O(m\sqrt{n})$ algorithm [HK73], where $m = nd$, $n =$ number of vertices and $d =$ degree of each vertex, exist for solving the maximum matching problem for regular bipartite graphs, which would satisfy our local objective. However, to satisfy both the local and global objectives, i.e., in order not to use UAVs that have more capabilities than required by any of the roles that are going to be assigned by the current manager role, we are interested in minimum cost in addition to maximum matching. Algorithms, such as the polynomial time Hun-

Find a function $f : R \rightarrow S$ that maximises U
where

$$U = \sum_{j=1}^{n} \sum_{l=1}^{k} U_{j,l}$$

$$U_{j,l} = \begin{cases} \sum_{i=1}^{m} u_i(r_j) w(u_i(r)) + \\ \sum_{i=1}^{p} u_i(s_l) w(u_i(s)) + \\ \sum_{i=1}^{q} u_i(r_j, s_l) w(u_i(r,s)) & u(s_l)u(r_j, s_l) > 0 \\ 0 & u(s_l)u(r_j, s_l) = 0 \end{cases}$$

Figure 4.13: The Role Assignment Problem

garian algorithm [Kuh55, Kuh56], that can compute the minimum-cost maximum matching for regular bipartite graphs exist. The role assignment bipartite graph can be easily transformed to a regular graph by adding dummy roles or UAVs if there are more UAVs than roles or vice versa respectively. The collocation constraint $colloc(r_i, r_j) = -1$ is satisfied by default because the bipartite matching model assigns each role to a unique UAV. We do not consider the $colloc(r_i, r_j) = 1$ constraint because roles that have to be placed together can be designed as tasks and placed in one role. However, should there be a need to consider it, our problem solving model is capable of dealing with this constraint because the constraint can be easily encoded into the matching problem by defining a third role r_k where $req_{cap}(r_k) = req_{cap}(r_i) \cup req_{cap}(r_j)$.

We specify the weight function, $weight(utility)$ for utilities using policies in order to make the optimisation system flexible and adaptable. The role assignment algorithm is shown in Algorithm 1. R_w is the set of roles waiting to be assigned and T_w is the waiting period before optimisation is started. The optimisation algorithm waits for a specified duration (T_w), after the first UAV is discovered, before it starts the optimisation. If this waiting period is zero, i.e., UAVs are assigned immediately as they are discovered, then the algorithm will be the same as the immediate role assignment algorithm. By waiting for a specified amount of time, it allows for discovery of multiple UAVs and hence enables the possibility of optimisation. A longer waiting period increases the probability of discovering more UAVs thereby increasing the probability of higher utilities at the cost of delaying the mission. In a similar manner to the weights of utility functions, the choice of the waiting period depends on the type of

the mission and should be decided by the mission administrator. Consequently, it is specified, using policies, as part of the mission specification.

Algorithm 1 Role Assignment Algorithm

Require: $R_w, T_w, c, weight(utility)\ denoted\ as\ w(u)$
Ensure: $f' : R' \rightarrow S' where R' \subseteq R, S' \subseteq S$
while $T < T_w$ **do**
if UAV s_l is discovered **then**
if $G = \emptyset$ **then**
Create G
for $r_j \in R_w$ **do**
if $u(s_l)u(r_j, s_l) > 0$ **then**
$cost = c - (\sum_{i=1}^{m} u_i(r_j)w(u_i(r)) + \sum_{i=1}^{p} u_i(s_l)w(u_i(s)) + \sum_{i=1}^{q} u_i(r_j, s_l)w(u_i(r, s)))$
else
$cost = c$
end if
Connect r_j to s_l with $cost$ as the weight of the edge
end for
end if
end if
end while
Transform G into a regular bipartite graph G'.
Revise the cost matrix of G'.
Create matching M from G'.
if M is not a perfect matching **then**
Compute $M =$ the minimum-cost maximum-matching of G'.
end if
Remove dummy vertices.
$f' =$ the resulting bipartite graph where,
$R' =$ vertices of the role bipartition with edges.
$S' =$ vertices of the UAV bipartition with edges.
$R_w = R_w \setminus R'$
return f'

Utility Models for Optimal Role Assignments

We use three utility functions for optimal role assignments, a role utility, a UAV (system) utility and role-UAV utility as shown in Figure 4.14. The role utility, $priority(r)$, measures the benefit of assigning role r with respect to the priority this role has compared to other roles. The UAV utility, $power(s)$, measures the benefit of using UAV s with respect to battery power compared to other UAVs. The $matching(r, s)$ utility measures the benefit of assigning role r to UAV s with respect to capability matching. The overall utility is computed by the $AggregateUtility$, which uses the weights provided through policies (Figure 4.16) for each utility. $AggregateUtility$ checks all utilities that are loaded onto the management system (Figure 4.15) identifies their

types and aggregates their values, thereby enabling the addition of new utility functions dynamically using policies.

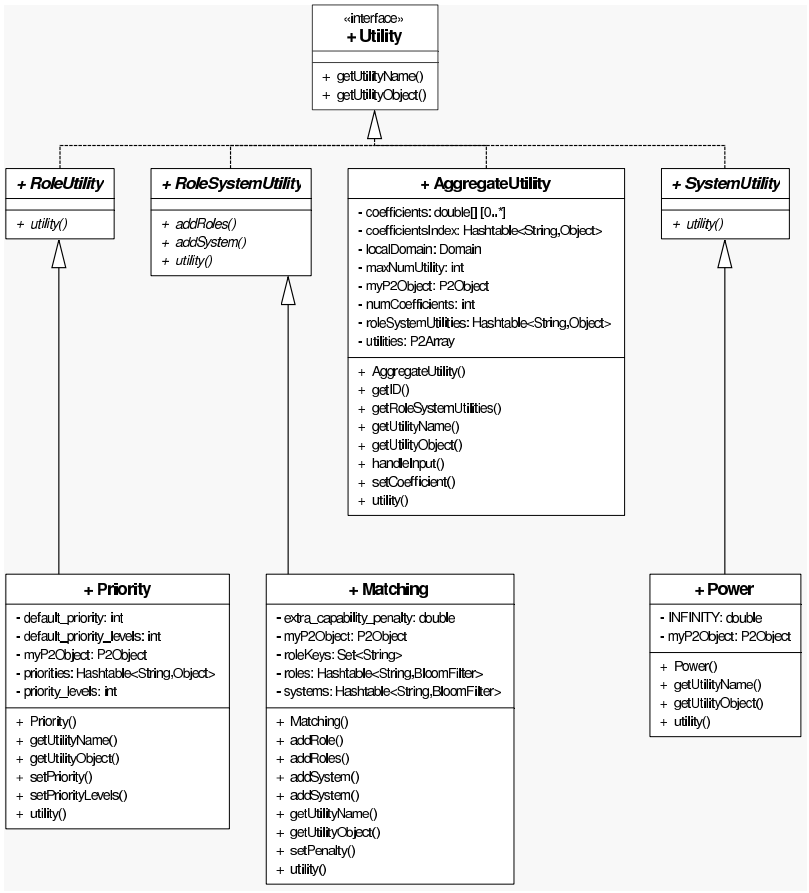


Figure 4.14: Utility Classes

Consider the reconnaissance team shown in Figure 4.6 again, with the required and provided capabilities shown in Figure 4.8 and with the UAV arrival order, $UAV_1 \rightarrow UAV_2 \rightarrow UAV_3 \rightarrow UAV_4$, as shown in Figure 4.9. We have previously shown that the team will never become complete with this arrival order. We will now show that using the optimal assignment algorithm the team will reach its optimal configuration.

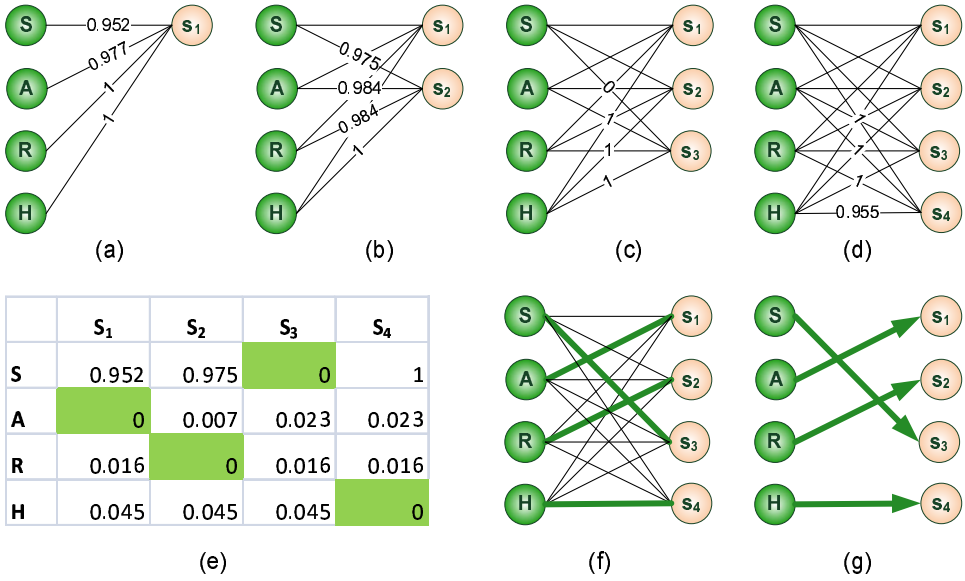
Figure 4.17 illustrates traces of the optimal assignment algorithm execution with the inputs of the algorithm $T_w =$ waiting time before optimisation = 1000 ms, $c =$ constant to transform utility into cost = 1, $weight(utility) =$ weight of a utility function, set

```
1 on assigned(role)
2 do loadUtility("Matching","Priority","Power","Aggregate")
3 when role("Commander") and optimise("true")
```

Figure 4.15: Utility Loading Policy

```
on assigned(role)
do setAggregateWeight("Matching",1,"Priority",0,"Power",0)
when role("Commander") and optimise("true")
```

Figure 4.16: Weight Policy



	S_1	S_2	S_3	S_4
S	0.952	0.975	0	1
A	0	0.007	0.023	0.023
R	0.016	0	0.016	0.016
H	0.045	0.045	0.045	0

Figure 4.17: Trace of the Assignment Algorithm

by the policies shown in Figures 4.16 & 4.18. As can be seen from the policy in Figure 4.16, for this example, the *power* and *priority* utilities are not included in the aggregate utility computation because their weight is set to zero, which makes the *matching* utility the deciding factor for assignment. In this example, the utility function used for matching is $utility(x) = \frac{1}{a*(1+x)}$ where $x = r \triangle s$ is the symmetric difference between role r's required capability and UAV s's provided capability. This and a variant of this function (shown in the case study chapter) are used to capture the utility of matching capabilities. Bloom filters [Blo70] are used to represent the capability descriptions; this enables efficient computation of set operations. Figure 4.19 shows the *matching* utility function for $a = 1$.

As shown in Figure 4.17(a), first when s_1 is discovered its *matching* utility is computed with each role waiting to be assigned. The same applies for UAVs s_2, s_3 and s_4,

```
on discoveryReady()
do setOptimRate("1000")
```

Figure 4.18: Optimisation Rate Policy

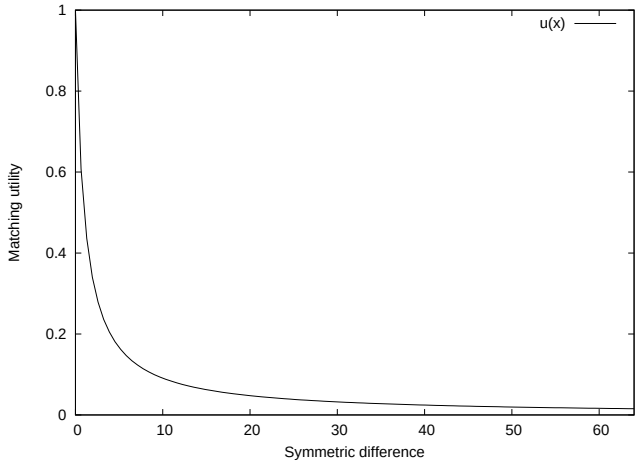


Figure 4.19: Capability Matching Utility Function

by which time the timer set by the policy shown in Figure 4.18 goes off and the optimisation starts. The resulting cost matrix[5] of the optimisation is shown in Figure 4.17(e) and the edges of the bipartite graph that have zero cost are marked as shown in Figure 4.17(f). The assignment will then take the edges with the lowest cost for each role as shown in Figure 4.17(g) resulting in an optimal configuration where every role is assigned to the best possible UAV.

4.3 Team Maintenance

The self-management framework maintains a team of hierarchically structured UAVs by means of the management tree. Both manager and managed roles send periodic state updates to each other, which are used for team maintenance. The content of the state update can range from only including information necessary to maintain the hierarchical management structure to mission specific operational information such as hazard location to remote role references used for maintaining collaboration structures.

4.3.1 Failure Management

In order to distinguish between intermittent communication link failures and permanent communication link or UAV node failures, different timeouts are used. Each

[5]In this example, the optimal assignment is found right after revising the cost matrix, i.e., before running the minimum-cost maximum-matching algorithm.

UAV periodically sends state information to its parent in the management tree; if the state information is not received within a specified timeout it is considered that a failure has occurred. The timeouts are: (a) T_C: detects intermittent communication link failure (b) T_N: detects permanent failures ($T_N > T_C$).

Failure of a communication link and/or a UAV causes partitioning of the team as well as loss of functionality. A systematically defined identity for UAVs is used to facilitate merging and re-joining of partitioned teams. The identity I of a UAV is defined as: I = $[M \mid H \mid S]$ where: M = mission ID, H = hierarchy level and S = a numbering system used to ensure that all the UAVs in the management tree can be placed in a total order. This identity lasts throughout the team configuration.

Adapting to Communication Link Failure

An intermittent communication link failure may be caused by either a temporary signal blockage by physical objects or movement out of communication range. Although local functions can keep operating, a temporary partitioning of the network over which the management tree is formed can cause disruption of state aggregation as well as the flow of management commands. In addition, remote operations will also be affected. The desired response to this type of failure is to continue mission execution with disconnected operations and resolve inconsistencies when the communication link reappears.

When the team is partitioned as a result of failure, one or more teams without commanders will be formed. In order to keep the mission executing during the failure, the top UAV in the hierarchy (which was already managing this sub-team during normal functioning) will become the commander of the team. A partitioned sub-team can also admit new UAVs. When the sub-team rejoins the parent team, the sub-team commander reports its current state to its parent and the domain structure of all UAVs in the mission is updated to indicate new members. To facilitate merging of partitioned teams, we define the hierarchy level of the partitioned team to be the level of its manager. Merging is performed by placing lower level hierarchy teams under the management of higher level hierarchy teams.

In this approach, there is no new role assignment or reassignment of existing UAVs to roles different from their original ones. Consequently, the mapping of existing

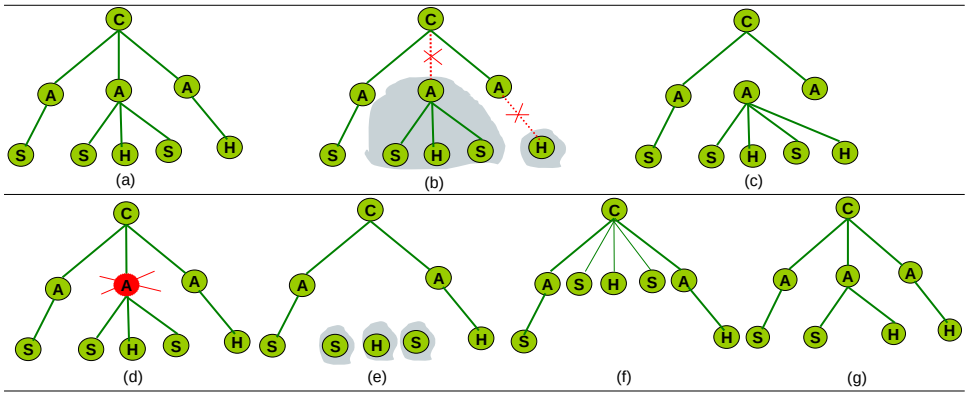


Figure 4.20: Reconfiguration and Role Reassignment to Adapt to Failure

UAVs to roles remains the same whereas the management tree can be different, as it is assumed that the adaptation is temporary. The initial configuration is shown in Figure 4.20(a). When communication link disconnection occurs, as shown in Figure 4.20(b), partitioned sub-teams are created. These sub-teams perform reconfiguration where the partitioned role, H, comes under the control of the other sub-team as shown in Figure 4.20(c).

Adapting to System Failure

A permanent failure is caused by either a node or communication link failure (other UAVs cannot distinguish between these). The result is the partitioning of the team as well as a loss of roles. The partitioning problem is addressed using the approach presented in Section 4.3.1. The response to the loss of roles is as follows (in order of priority): (i) use replicated roles, if available, (ii) if there are unassigned or newly discovered UAVs, perform a role reassignment, while keeping the existing team configuration, to replace the lost role(s), and (iii) if none of the above is feasible, reconfigure the team by swapping less crucial roles for more crucial roles. Should the reconfiguration incur role replacement, this takes place only in subsets of the team that are lower in the hierarchy than the failed UAV. This is due to the fact that roles assigned to higher level UAVs are assumed to be more crucial to the mission. In the case of role reassignment and reconfiguration, state information migration takes place.

Figure 4.20 illustrates adaptation to permanent failures. The initial configuration is shown in Figure 4.20(a). When a permanent failure occurs, as shown in Figure 4.20(d), partitioned sub-teams are created (Figure 4.20(e)). The response to this prob-

lem can be either reconfiguration as shown in Figure 4.20(f), where the partitioned sub-teams are moved up in the management hierarchy and now managed by the main commander, or a role replacement where the UAV that was previously assigned to role S is now reassigned to the supposedly crucial role A as shown in 4.20(g). All reconfigurations, reassignments and other responses are specified in terms of policies.

4.4 Comparison with Related Work

Research in multi-robot and multi-agent systems has resulted in a number of approaches for dynamic team formation and role (task) assignment. In this section, we compare our work to related work in this area. For a more complete description of the related work, the background chapter can be consulted. Also, note that the mission specification aspect of the related work, where applicable, is considered in Chapter 3.

CDL [MAC97] takes the difference in capabilities of robots into consideration in that behaviours are assigned to robots based on their capabilities. It is similar to our approach in this aspect. However, it forms and maintains a team by using a static matching and allocation of behaviours to robots and hence has no discovery and optimisation elements. The matching and allocation is performed by the mission administrator. CHARON [AGH$^+$00] uses a similar static allocation but has no matching element as it does not take the difference in capabilities of robots into consideration.

ALLIANCE [Par98], initially, forms a team in a static manner where the assignment of behaviours to robots, based on their capabilities, is done by the administrator. Our approach forms the initial team dynamically through discovery, optimisation and capability matching. Once formed statically, ALLIANCE's team can, however, maintain and optimise itself through time by using dynamic task reassignment that is facilitated by the broadcast of activities by each robot. Upon receiving the broadcast, each robot decides whether to take a task away from the current robot performing that task based on the motivational level computed using the received information. AYYLU [Wer00] uses a similar static initial team formation with broadcast-based task reassignment. The dynamic reassignment of tasks (behaviours) in both frameworks

applies only to existing members of the team since there is no means to introduce new team members dynamically. In contrast, our framework allows for admission of new UAVs in order to replace failed UAVs but does not perform re-optimisation. Re-optimisation of all the role assignments either periodically, after a failure or when a new node is discovered was considered. This would result in dynamic role reassignment, but we decided the cost of transferring roles, policies and state information outweighed the benefits.

MURDOCH [GM01, GM02] uses auctions and contracts [Smi80, DS83] to form and maintain a team dynamically. Its task allocation approach is completely distributed as opposed to our hierarchical approach. Consequently, MURDOCH's task assignment decisions are based on the current and/or local situation only without taking into account how the decision might affect the future and/or the global situation. In effect, its task allocation approach may not always give the best solution. Our approach takes the global and/or future situation into consideration. In the window of time that every role assignment is done, the architecture performs optimisation on the set of discovered UAVs and roles waiting to be assigned.

4.5 Conclusion

In this chapter, we have presented a distributed, dynamic team formation approach that discovers UAVs and optimally assigns them to roles based on their capabilities. The approach also caters for intermittent communication link disconnection and permanent link or UAV failures by using policies to adapt to failure.

The framework is novel in that it starts with a single mission specification defining the overall mission policies and distributes roles in an optimal manner as UAVs are discovered. The overall management of team formation and maintenance of dynamic teams is distributed among members of the team and hence is scalable and not dependent on any single node. The policy-based local management decisions facilitate adaptation to local context.

Automating the process of discovery and optimal role-to-UAV allocation is crucial, for dynamic team formation, in applications where the types of UAVs are not known in advance. Similarly, the ability to maintain the team is necessary in teams of

mobile autonomous systems where communication link or node failures can happen frequently. In addition to enabling these abilities, the prominent advantage of our approach is that instead of encoding all the decisions in the framework, we factor out management decisions that may vary across missions and allow for adapting them through policies.

Although the initial role assignment is done in an optimal manner, the framework does not consider re-optimisation in order not to destabilise the team. However, in cases where the re-optimisation benefit outweighs the cost, it would be useful to allow this behaviour by specifying optimisation policies. For example, when role reassignment is performed to adapt to failures a global optimisation could be considered instead of the much localised optimisation that considers only the failed roles.

Chapter 5

Communication Management

UAVs are equipped with wireless transceivers that would enable them to form a multi hop mobile ad hoc network (MANET) . As mentioned in Chapter 1, communication between any two UAVs in a team can be achieved by means of a MANET routing protocol so long as there is a wireless network connectivity among team members. However, given a set of UAVs when a UAV moves far enough from the rest of the UAVs to an extent that it is not within the communication range of any of the other UAVs, the UAV becomes disconnected from the rest of the network. Thus the MANET routing protocol will not be able to provide full connectivity between all UAVs. The impact of a UAV's movement out of the communication range could result in its isolation or it might even result in partitioning of the team if it is the sole relay path within a part of the network. Connectivity can be maintained by enforcing strict formation control, such as a geometric formation control [SY99], that would keep the original network structure (as defined at mission startup) intact. This, however, would render the team dysfunctional, for most but strongly coupled missions such as coordinated transport of objects, as the formation control has to override the mission requirements (i.e., requirements that would need the UAV to stay at a certain location) all the time.

One, more general, way of addressing this issue is introducing a mission-specific *Relay* UAV (UAV enacting a *Relay* role) whose sole purpose is extending the communication range of its manager role (UAV enacting the manager role that could be any type of role depending on the mission specification) thereby providing more freedom

of movement to its manager role in particular and all the other team members in general. Consider the reconnaissance team (Figure 4.6) at a certain arbitrary time with the UAVs being in the positions shown as in Figure 5.1 (a). Figure 5.1(b) shows the network connectivity graph (broken lines) and the management tree (solid lines) for this team. In this figure, the UAV enacting the *Hazard-detector* role (UAV_8), which is managed by the *Aggregator*, is at a distance that is greater than the communication range of the UAV enacting the *Aggregator* role (UAV_1). However, it is still connected to the rest of the team members because of the presence of the *Relay* UAV (UAV_5).

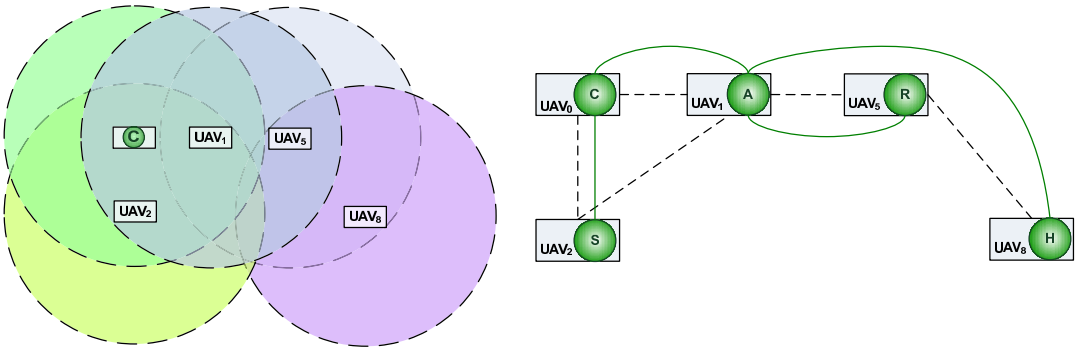


Figure 5.1: Network Connectivity

Ideally, a mission can enhance its network connectivity by employing more *Relay* roles thereby effectively creating a backbone network made up of relay nodes, which are not constrained by mission requirements (i.e., other than the requirement to stay within the communication range of their manager which gives more freedom than other requirements that would need a UAV to stay at a certain location). This solution, although feasible, is costly since if the mission has to solely depend on dedicated *Relay* roles for communication link maintenance then a number of *Relay* roles (depending on the size of the mission area, the communication range of the UAVs and the number of UAVs in the team) are required. We can, however, augment this solution by (1) requiring all UAVs to move in a manner that would maintain the connectivity whenever they are not constrained by their mission requirement, and (2) enforcing a rendezvous of all UAVs at a specified location, when disconnection is imminent, in order to create intermittent connectivity. Our framework addresses this issue by using two complementary measures. The first one is a proactive approach where the framework tries to prevent UAVs from becoming disconnected from the network and the second one is a reactive approach, triggered by imminent loss

of communication link among UAVs, where the framework tries to setup rendezvous in order to enable communication at intermittent intervals. Both the movement and rendezvous-based approaches do not necessitate the presence of UAVs dedicated to performing *Relay* roles since UAVs in the mission can potentially perform the relay function to maintain an ad hoc network by modifying their path of movement to maintain communication range with neighbours while still performing their other roles. However, having some nodes dedicated to relay roles can improve communication maintenance if most of the other roles do not have the flexibility in adjusting movement patterns.

5.1 Maintaining Communication Links

The communication layer of the framework deals with communication link maintenance by using two different approaches. Each approach is more suitable to distinct types of missions or classes of missions. In the first approach, UAVs try to control their movement so as to make sure that they stay within the communication range 3 (Section 5.1.1). Consequently, this approach is suitable for teams of UAVs that operate in a convoy manner. In the second approach (Section 5.1.2), instead of restricting the motion of the UAVs, UAVs perform disconnected individual operations while maintaining the team structure by trying to ensure that all members of the mission, regardless of destination or role, communicate at intermittent intervals. If the second approach fails, the issue will go up to the failure management protocol in the team layer. For missions or contexts that result in UAVs operating in a scattered manner, the second (i.e., rendezvous-based) approach is more suitable as it does not restrict the movement of UAVs. In this work, we use both approaches by starting with the first approach and switching to the second approach when the need arises. We do so in order to accommodate different types of missions without losing the advantage of each approach for specific types of missions. However, either one of the algorithm can be disabled if the behaviour of the mission is known a priori.

In the following, we motivate the need for both types of approaches and then present the algorithms. Consider a reconnaissance team that consists of a *Commander*, a *Surveyor* and a *Hazard-detector* roles. The team members as a whole may effectively move as one entity since every next point explored in the mission area is decided

by the *Surveyor* with the other roles facilitating the process by providing functionalities that are not available in the *Surveyor* UAV (e.g., hazardous material detection by the *Hazard-detector* UAV, and high-bandwidth long-range communication by the *Commander* UAV). The first approach can then be used to maintain communication among these UAVs, which are operating in convoy, by intermediate UAVs acting as communication relays to the furthest ones. In this scenario, the team was able to operate in convoy because the cooperation is based on functional partitioning (i.e., UAVs with different capabilities working near to each other at all time since each capability is needed at every location they work on) as opposed to spatial partitioning (i.e., UAVs with identical capabilities partitioning the mission area and performing the reconnaissance accordingly). Although this approach allows UAVs involved in a mission to maintain communication links and is suitable for teams operating as a convoy, it does so by constraining the movement of UAVs. Consequently, it would not be feasible (conflicts with mission requirements) in applications such as large area reconnaissance where a larger number of UAVs need to operate in a scattered and less constrained manner due to the partitioning of the mission area. The second approach is suitable to this type of missions. It is also possible, in both cases, to use dedicated mission-specific relay (repeater) UAVs whose sole purpose is relaying (extending the communication range) of other UAVs in the mission in order to maintain communication link among UAVs. However, as mentioned before, solely depending on this solution is costly as it increases the number of UAVs required to perform a mission.

Before we discuss the algorithms and protocols involved in the communication management scheme, we will list the assumptions:

- Each UAV knows its current location (e.g., by means of a GPS device) and also its direction and speed of travel.
- No clock synchronisation is assumed (although the clock rate on the UAVs is assumed to be nearly equal, for example, 20 minutes on one UAV is approximately equal to 20 minutes on another).
- All UAVs have the same communication range (C_R) and,
- A global/local co-ordinate system exists for specifying location and direction of travel.

- UAVs move on a plane and the dimensions of UAVs, when projected on to this plane, is negligible compared to the communication range.

Consider again the UAV management tree shown in Figure 4.6. Each UAV periodically reports its state information to its parent node and vice versa. Since this periodic message contains the current location, direction of travel and speed of the UAV, it allows UAVs to monitor the current position of each other. We will discuss the two approaches in the next two sections.

5.1.1 Adapt Movement to Maintain Communication

In this section, we detail the approach that controls the movement of the UAVs to ensure that they maintain communication with all member of the team. In this approach, one of the UAVs is the leader and thereby dictates the direction of movement. All the other UAVs follow this UAV by using the following algorithm:

1. Receive state update, about other UAVs in the team, containing location, speed and direction of travel.
2. After T amount of time, predict the distance between the leader UAV and each non-leader UAV.
3. If the distance between self (the UAV running this algorithm) and the leader UAV is the smallest then set the target UAV to be the leader UAV, otherwise compute the distance between self and each non-leader UAV and set the target UAV to be the nearest UAV to both self and the leader UAV.
4. If the distance between self and the target UAV is greater than the communication range threshold (modelled as a percentage of the communication range), move to target.

In the following, we illustrate how this algorithm works. Assume that the position at time T of the 5 UAVs listed in the management tree in Figure 4.6, are as shown in Figure 5.2 (a). Starting at time T, UAV S starts to move from its current location to its future location S' with constant speed and direction (θ_S). Since the direction and speed of S are available to the rest of the UAVs in the team, they can predict

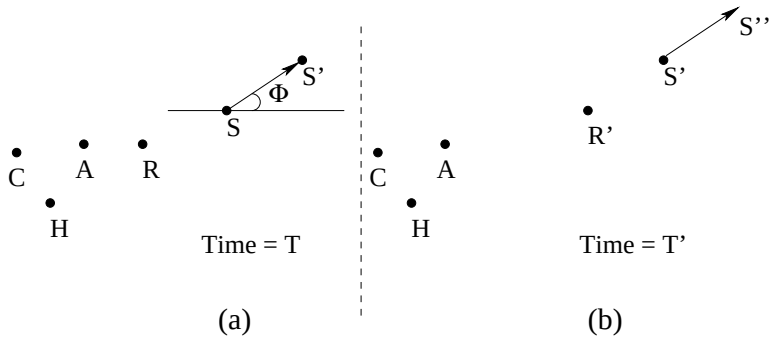


Figure 5.2: Position of UAVs

the location of S at a later time (T'). If this position is beyond the communication range of the rest of the UAVs in the mission, the closest UAV to S starts to move so as to make sure that it still is within communication range of S. As per the scenario mentioned above, we can see from Figure 5.2 (a) that UAV R is the closest to UAV S and it is R's responsibility to make sure S is within communication range and so it moves accordingly. When S moves to S' at time T', R moves to R' (Figure 5.2 (b)). The amount that R has to move depends on its location and the location of S. By time T', R moves in a straight line to R', which is the closest point to its current location that is within communication range of S'.

If UAV S keeps moving in the same direction and moves from position S' to S'' during the next time period, then R would also move to keep S within communication range, provided it does not lose communication with the rest of the group. In the event that R along with S move out of communication range with respect to the rest of the UAVs in the group, the UAV closest to R will start following R to keep it within communication range. If S keeps moving away, the rest of the UAVs form a "chain" that allows them to maintain communication with S. Ideally, the algorithm enables the UAVs to move in a convoy fashion while maintaining communication link among themselves. However, even in a convoy type mission a UAV may not always be able to move when it should do so due to (1) mission requirements (e.g., a *Hazard-detector* may not have finished its detection process and hence cannot follow the *Surveyor* in time although it is the nearest to the *Surveyor* and must have followed it immediately in order to keep the communication link) (2) mission parameters – either the communication range threshold not being small enough and/or the state update not being frequent enough in order to trigger the follower movement in time. Hence, by the time the movement is triggered the target may have travelled too far thereby rendering the

link maintenance algorithm ineffective.

In the case of the conflict with mission requirement, we have made a design choice of giving priority to the mission tasks thereby allowing UAVs to decide whether to follow the leader or any other target UAV depending on whether they need to stay at their current position or not. Although this choice incurs an occasional breakdown of communication link among UAVs, this problem can be reduced by switching to the second (rendezvous-based) approach when the movement-based communication link maintenance approach does not work.

The choice of the communication range threshold and state update rate have considerable impact on the success or failure of the algorithm. The effect of these parameters is evaluated and the result is presented in Chapter 8. The algorithm does not introduce additional messages since it gets the location, speed and direction information from state update messages that are already used to maintain the management tree. However, it increases the size of these messages by introducing additional states, i.e., location, speed and direction, and thereby introduces a bandwidth usage overhead (discussed later in Chapter 8).

For a team consisting of n UAVs, the computational overhead for the UAV that is nearest to the leader is $O(n)$ since it has to perform $n-1$ distance estimation computations in order to decide the nearest UAV to the leader. For all other follower UAVs, the computational overhead is $O(n^2)$ since they have to perform $n-1$ distance estimation computations in order to decide the nearest UAV to the leader followed by $(n'-1)n'/2$ computations to decide the nearest UAV to themselves, where $n' = n-1$, i.e., the total number of UAVs less the leader UAV.

5.1.2 Rendezvous to Restore Communication

In this section, we will detail the approach that allows UAVs to perform disconnected individual operations, while maintaining the team structure by trying to ensure that all members of the mission regardless of destination or task, communicate at intermittent intervals.

Consider again the UAV management tree shown in Figure 4.6. If the *Commander* UAV notices that the distance between a child node and another member is greater

than or equal to the *range threshold* (T_R, which is modelled as a percentage of the communication range (C_R)), it initiates the rendezvous algorithm (Algorithm 3). Using the current location, speed and direction of the UAVs in the mission, Algorithm 2 calculates a *rendezvous area* where all the UAVs are expected to rendezvous after a specified time. Once an instance of the rendezvous algorithm is running, additional requests are ignored, since the rendezvous area has already been calculated and it is assumed that the newly departing UAV will eventually rendezvous at the same area. After reaching the rendezvous area, the algorithm is restarted only if the need arises again.

Algorithm 2 Calculate Rendezvous Area

Require: L, V, A
Ensure: *REND_AREA*

1: θ = $\arctan(\frac{v_1 sin\theta_1 + v_2 sin\theta_2 + ... + v_n sin\theta_n}{v_1 cos\theta_1 + v_2 cos\theta_2 + ... + v_n cos\theta_n})$
2: (X, Y) = $\begin{cases} X = \frac{x_1 + x_2 + ... + x_n}{n} \\ Y = \frac{y_1 + y_2 + ... + y_n}{n} \end{cases}$
3: $D = T * v_{min}$
4: (X_{RP}, Y_{RP}) = $\begin{cases} X_{RP} = X + D * cos\theta \\ Y_{RP} = Y + D * sin\theta \end{cases}$
5: Calculate *REND_AREA* based on *REND_PT* = (X_{RP}, Y_{RP})
6: **return *REND_AREA***

Algorithm 3 Rendezvous Algorithm

1: Broadcast *REND_MSG*
2: Calculate rendezvous area
3: Send the *REND_AREA* and T to the team members
4: **return**

The rendezvous area is calculated as follows (shown in Algorithm 2 and used by Algorithm 3). The direction of travel is calculated by computing the resultant direction of travel of all the UAVs in the mission with respect to a common axis. Once the direction is calculated, the *rendezvous area* is calculated to be the area surrounding the *rendezvous point* that is achieved by projecting the speed of the slowest UAV starting from the average location onto the direction of travel over the requested time (T). The notations used in the two algorithms are:

- *REND_MSG*: requests the UAVs to send their current location, speed and direction,
- n: number of member UAVs that reply to *REND_MSG*,

- L: list containing the location of the UAVs, $\{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$,
- V: list containing the speed of the UAVs, $\{v_1, v_2, ..., v_n\}$,
- A: list containing the direction of the UAVs, $\{\theta_1, \theta_2, ..., \theta_n\}$,
- v_{min}: minimum speed from among $\{v_1, v_2, ..., v_n\}$,
- θ: resultant angle of the direction of travel,
- (X, Y): average location,
- T: time to rendezvous (this time is relative to current time and indicates the time in the future when the nodes need to rendezvous),
- (X_{RP}, Y_{RP}): rendezvous point,
- *REND_AREA*: rendezvous area - a suitable expression for an area around *REND_PT*, and
- D: distance to rendezvous.

The rendezvous algorithm (Algorithm 3) is initiated by the *Commander* UAV when it becomes aware that a member UAV is about to go out of range. This range (referred to as *range threshold* and denoted by T_R) has a direct bearing on the protocol. T_R must be less than or equal to the communication range of the UAVs (C_R) due to the fact that the departing UAV (i.e., the UAV that is the cause of the rendezvous set up) must receive information about the *rendezvous area*. Since the communication range is a parameter that is known, T_R can be modelled as a function of the communication range: $T_R = \sigma * C_R$, where $0 < \sigma \leq 1$. σ is an adaptable parameter that can be changed depending on the mobility of the UAVs involved in the mission. If the UAVs involved in the mission are more or less stationary and their speeds are low, then σ can be a high value (closer to 1) since the algorithm can compute and disseminate the *rendezvous area* before the UAV moves out of communication range. If the mobility and speed of the UAVs is high, the value of σ should be low (lower range threshold). This would trigger the rendezvous algorithm earlier, thus ensuring that the *rendezvous area* is calculated and disseminated before the UAV moves out of communication range. The mobility behaviour of the UAVs in the mission depends on the type of the mission and hence σ is set by the mission administrator.

The *Rendezvous Area* is the area surrounding the *rendezvous point* (P) within which UAVs must re-gather after a certain time (T) to re-establish communication. We draw the rendezvous area as a circle (with radius R) that contains the rendezvous point. There are three choices for drawing this circle (as shown in Figure 5.3).

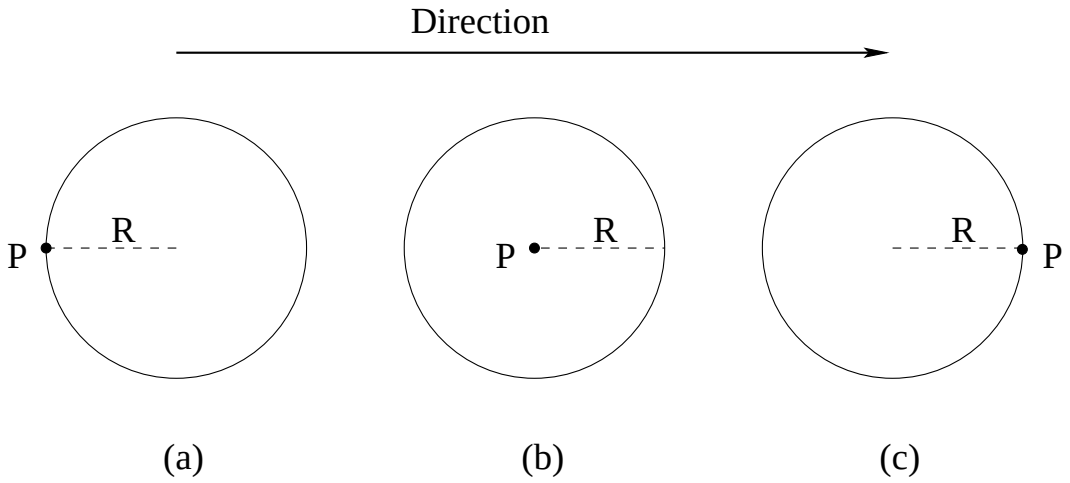


Figure 5.3: Drawing the Rendezvous Area around the Rendezvous Point

In Figure 5.3 (a), the rendezvous area is drawn by having P on the circumference of the circle that denotes the rendezvous area with the area itself stretching in the direction of travel. This is the most optimistic solution wherein it is assumed that the slowest UAV (with speed = v_{min}) manages to speed up to reach P. This is a valid assumption since it is quite likely that the UAV that is the slowest at the time when information was exchanged for the execution of the rendezvous algorithm is not travelling at its maximum speed.

In Figure 5.3 (b), the rendezvous area is drawn by having P as the middle point of the area with the area as a circle of radius R around P. This is a less optimistic solution as compared to the one in Figure 5.3 (a). In this scenario, there is a higher probability of the slowest UAV reaching the area.

In Figure 5.3 (c), the rendezvous area is drawn by having P on the circumference of the circle that denotes the rendezvous area. This circle is drawn with P as an end point in the direction of travel. This is the most pessimistic solution of the three mentioned in Figure 5.3. This solution assumes that the slowest UAV does not increase its rate of speed and tries to ensure that the slowest UAV does indeed reach the area.

Given the mission specifications and an idea about the terrain and area, it may be possible to devise other mechanisms by which the area can be calculated. For example, if it is known that the UAVs are being deployed within a building and the map of the building is available, the rendezvous area could be specified with respect to rooms within that building.

The time to rendezvous (T) is used in Algorithm 3 for calculating the rendezvous point and consequently the rendezvous area. In Section 4.3.1, two timeout parameters, namely T_C and T_N, corresponding to intermittent link disconnection and permanent failure respectively were introduced. UAVs that are out of communication range for longer than T_N are assumed to have failed and may be replaced by other UAVs. Consequently, the rendezvous algorithm should decide on the rendezvous area such that the UAVs can converge on that area before T_N amount of time has passed, i.e., $T < T_N$.

Some of the assumptions we have made introduce limitations to both algorithms when applied in real-life missions. The assumption that all UAVs have the same communication range will not hold in real-life thereby causing the need to use the smallest communication range from the list of communication ranges corresponding to each UAV. Also, the assumption about the knowledge of current location limits the applicability of the algorithms in indoor missions where GPS reception is not reliable.

5.2 Comparison with Related Work

Multi-robot and multi-agent systems require communication in order to perform cooperative missions. In this section, we compare our work to related work in this area. For a more complete description of the related work [AGH+00, CKC04], the background chapter can be consulted.

CHARON [AGH+00] has a formation control system that enables a leader robot to move along prescribed trajectory while follower robots move with a pre-specified separation and bearing with respect to the leader robot. Similarly, the hybrid-automata [Hen96] based paradigm for cooperating robots presented in [CKC04] has a leader-follower type formation control that is based on broadcast of the position and velocity of the leader to all the followers. Although the objective of these formation control

mechanisms is not communication link maintenance, we observe that they can be used for this purpose provided that the separations are chosen in accordance with the communication range of the robots. However, the algorithms are designed for tightly coupled tasks such as cooperative transport and manipulation of objects and consequently try to keep fixed separation among robots at all time. This results in synchronised movement and hence will not allow one UAV to perform a stationary task (e.g., hazard detection) while others are moving. In contrast, our movement-based link maintenance algorithm allows this type of behaviour provided that the movement patterns of the leader or target UAV do not result in them moving out of communication range.

In [AOSY99], the authors presented a distributed algorithm that allows autonomous mobile robots with limited visibility (robots can only detect the presence and estimate the distance of a subset of all the robots in the mission) to converge to a single point (the robots are assumed to be point size and collision is neglected). The algorithm allows the robots to calculate their next position based on only the location information of their neighbours (those that are visible); then move for a specified distance and direction and then repeat the whole process until the robots converge to a single point. The next position is calculated in such a way that the new region (a circle) containing the robot and its neighbours is smaller than the current region. This is achieved by computing, for the robot and all its neighbours, the maximum distance each can travel towards a common (central) point without leaving the region, selecting the minimum distance out of the set of results, and then restricting the distance a robot moves in every step to be less than this value. The common point itself may also change in every step of the algorithm. Similarly, the authors in [Lin05, LMA07] address the problem of convergence to a single location of a group of mobile autonomous agents. The authors employ two different strategies (with and without common clock) that allow for mobile autonomous agents to rendezvous at a single specified location. Both of the two approaches ([AOSY99, Lin05, LMA07]) are similar to our *rendezvous algorithm* in that they set up rendezvous. However, there are significant differences, mainly caused by their dependence only on "local" knowledge, wherein each agent independently calculates its new location based only on its observation of neighbour information without any communication. This, local-strategy based, approach is more robust in that the robots will always converge provided that

there is no hardware failure. Since the rendezvous (common) point can change with time, the less capable robots can effectively pull this point towards themselves in contrast to our approach where the point is fixed and robots that could not make it to this point, due to change in circumstances after the point is agreed, will be left out. However, in contrast to our approach, the local-strategy based algorithms do not have an upper bound of rendezvous time, which is a drawback for time sensitive applications. In addition, because the robots have to recompute the rendezvous point after each step, these approaches have more computational overhead than ours. On the other hand, these approaches have no communication overhead since the robots sense neighbouring robots and estimate the distance. In a similar manner that our approach needs a reliable positioning device, these approaches need a reliable distance (range) estimation devices such as sonar or infra red which are more costly than GPS devices.

5.3 Conclusion

In this chapter, we have presented two complimentary approaches that try to prevent communication link disruption among UAVs and setup a rendezvous to create intermittent communication interval if the prevention fails.

Our approach is novel in that we address communication link maintenance as part of the self-management architecture in contrast to multi-robot and multi-agent architectures that assume the existence of communication links throughout the mission execution time, or immediately revert to failure management when the communication link fails.

A priority-based approach is used to resolve mission requirement and communication link maintenance requirement conflicts. In this work, we give priority to the mission by default. A better way would be using a utility-function based approach. We have factored out parameters that affect the behaviour of the communication management in order to allow setting them through the mission specification and adapting them through policies provided that suitable utility models to estimate these parameters in accordance with the behaviour of the mission and characteristics of the mission environment are developed.

Chapter 6

Case Study: Search and Rescue

6.1 Introduction

In this chapter, we consider a search and rescue mission scenario to exemplify our self-management framework. Using this mission, we illustrate: (1) how a mission is specified in terms of roles & policies and how the specification for one type of mission can be reused for a similar or another type of mission, (2) policy-based dynamic team formation in accordance with the mission specification, and (3) policy-based mission adaptation.

As stated in Chapter 3, mission specification is performed by the mission administrator using the mission, role and policy specification approaches discussed in Chapter 3. The mission administrator specifies a mission in terms of roles that are themselves specified in terms of their interfaces and policies. In order to perform this specification, the administrator needs to decide on what type of roles to use, how to organise these roles (i.e., decide on the team structure) and what policies to use in order to direct the team into achieving the goal of the mission. These issues involve the manual process of refining the mission statement stated in a natural language in order to identify roles and their interactions, possible organisation of roles, and policies that dictate the roles' behaviours so as to achieve the goals of the mission. We envisage a mission designer performing this process for a class of missions resulting in a repertoire of role types, policies and role organisation structures that could be made readily available for the mission administrator to facilitate a faster and easier mis-

sion specification. Although refinement is not our framework's concern, for the sake of completeness and also to show that the example mission structure is based on a sound design, we present the refinement process followed by elements that relate to the framework.

The search and rescue scenario is loosely based on the urban reconnaissance scenario presented in [AC05] and the disaster scenario presented in [WR+04]. It was elaborated using various examples of search and rescue in the literature [DML03, SN00, Sch05, Mur04a, RKM+07].

Search and rescue missions have long been undertaken by teams of trained personnel around the world giving rise to the development of systematic organisation to create search and rescue teams. Figure 6.1 shows the organisation for the United States federal urban search and rescue task force [WR+04]. We use this task force organisation as an aid for identifying management relations between roles as we manually refine the search and rescue mission.

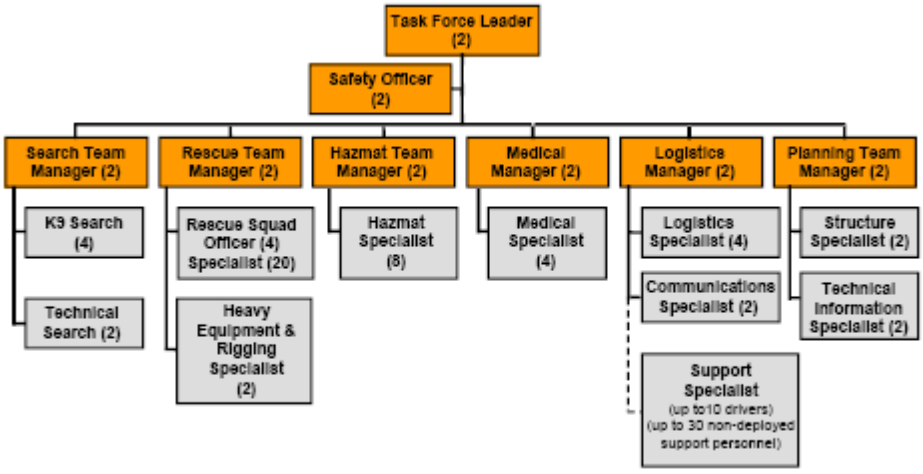


Figure 6.1: Urban Search and Rescue Task Force Organisation [WR+04]

6.2 Scenario

A residential town was hit by an earthquake that has a magnitude of 7 on the Richter scale leading to the collapse of most of the buildings in the town. Residential complex Alpha was one of the seriously damaged areas. This complex had ten 4-story buildings constructed on a one kilometre square area. The approximate number of

residents is 500. Although some residents have managed to evacuate the complex during the earthquake there are still about 300 residents that remain unaccounted. Electrical power service has been interrupted and all electronic communication infrastructure including fixed and mobile telephone networks as well as wireless and other Internet access channels have been destroyed.

The local observatory predicts that there will be several aftershocks within the first 24 hours of which some may have a high enough magnitude to cause more collapse. The local health and safety office warns that biological and chemical substances from a research laboratory of a collapsed hospital that was adjacent to the complex might contaminate the area.

6.3 Search and Rescue Mission

The mission statement provided to the mission command centre is as follows:

- Rescue survivors, and if needed provide them with emergency medical assistance before transporting them to a care centre. Information from the damage assessment and hazardous material detection should be used to assist the rescue activity. For example, priority should be given to survivors in a contaminated area. On the other hand, if the intensity of the hazardous material is high enough to significantly decrease the chances of survival of an identified survivor until rescued, priority should be given to survivors with a higher chance of staying alive until rescued.
- Assess and report the damage caused by the earthquake. A visual documentation of the damage has to be compiled and reported for later use. Analysis of this information as it is being collected has to be performed in order to detect and report potential collapses that might happen during the mission. Information about likely collapses should be used to prioritise rescue operations.
- Identify and report areas contaminated by hazardous materials. The areas should be marked on a map and the types of hazardous materials should be indicated.

- Ensure the presence of communication link among team members in order to report the progress of the mission to the command centre and keep all members of the team up to date with information that concerns them.

This mission is best assisted by UAVs for various reasons including the following:

- The mission can be dangerous to humans as structures that have not yet collapsed might collapse during the search and also the hazardous material may contaminate them.
- Humans may not be able to see and get through small gaps in the debris.

6.4 Mission Specification

We call the process of generating a mission class specification from mission statements a *mission refinement*. We use a multi-agent systems engineering approach [DWS01] to identify goals, structure them hierarchically and extract roles from the goal hierarchy. The following goals can be identified from the mission statements stated in Section 6.3:

1. Rescue survivors.
2. Provide emergency medical assistance to survivors.
3. Assess and report damage.
4. Identify and report areas contaminated by hazardous materials.
5. Ensure communication link among team members.

Now that we have captured both functional (1 - 4) and non-functional (5) goals, we can structure them into a goal hierarchy, which is an iterative process involving composition or decomposition of goals. As a first step we compose all the goals and form an overall system goal, namely *Manage earthquake disaster* to create the hierarchy shown in Figure 6.2.

We now decompose the goals continuously until we reach a point where further decomposition would lead to a specific way of achieving a goal; at that point we shall

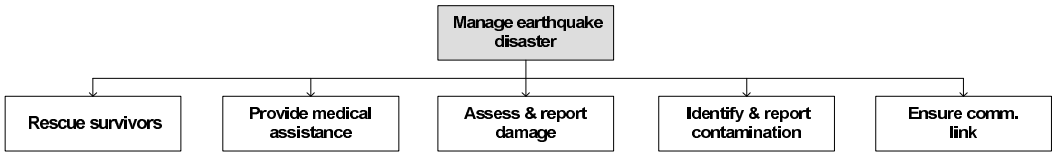


Figure 6.2: Initial Goal Hierarchy

stop the decomposition because we do not want to bind the role and consequently the UAV enacting the role to a specific means of achieving a goal. Figure 6.3 shows the goal hierarchy after a sufficient amount of decomposition.

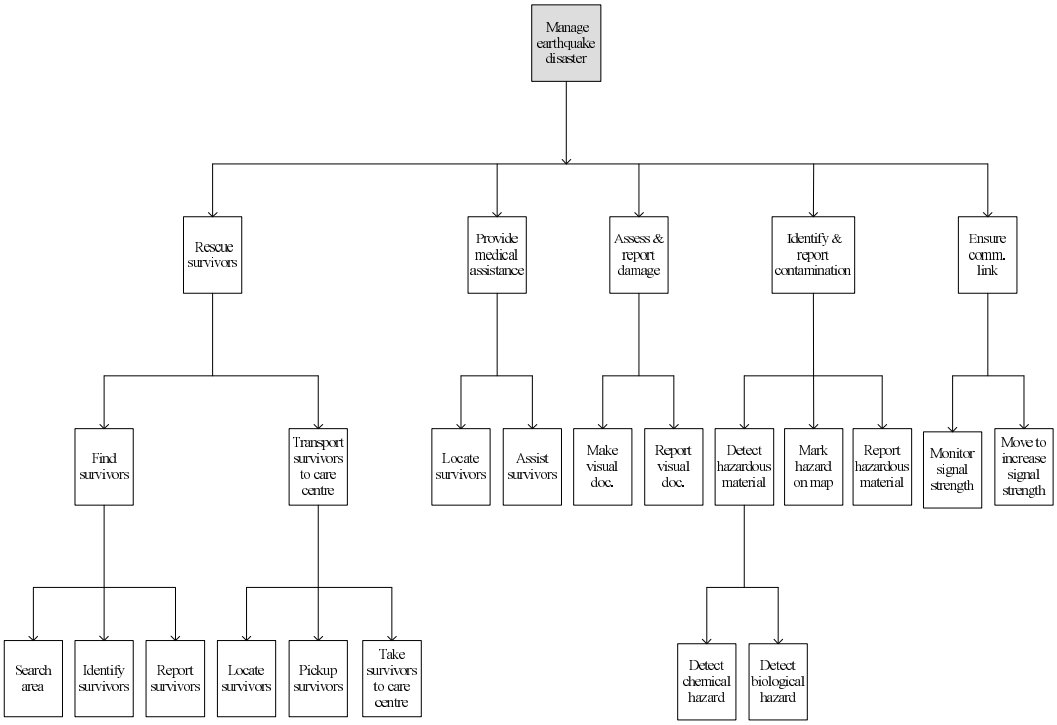


Figure 6.3: Goal Decomposition

The next step is identifying roles and outlining the interaction between the roles using possible events that might occur in the search and rescue system. Each identified role should satisfy one or more of the subgoals in the goal hierarchy. Using the goal hierarchy, shown in Figure 6.3, we add roles to our system until we have sufficient roles to satisfy the overall goal of the system. If we add a *Surveyor (S)* role that satisfies the *Find survivors* and *Assess & report damage* subgoals, a *Tranpsorter (T)* role that satisfies the *Transport survivors to a care centre* subgoal, then we satisfy the *Rescue survivors* subgoal. Note that we could have added a role that satisfies both subgoals and consequently satisfies the *Rescue Survivors* subgoal; the use of two roles instead is a design choice driven by an intention to separate duty and

thereby increase efficiency and minimise risk. On the other hand, in addition to partially satisfying the *Rescue Survivors* subgoal the *Surveyor* role also satisfies the *Assess & report damage* subgoal. Again, the use of one role to satisfy two subgoals is a design choice motivated by efficiency since the *Surveyor* role can perform both searching and visual documentation (e.g., taking pictures) simultaneously without compromising any of the subgoals. A *Medic (M)*, *Hdetector (H)* (stands for hazard-detector), and *Relay (R)* roles are added to satisfy the *Provide medical assistance*, *Identify & report contamination* and *Ensure communication link* subgoals respectively leading to the satisfaction of the overall system goal, i.e., *Manage earthquake disaster*. A *Commander (C)* role and an *Aggregator (A)* role are added to control the mission and facilitate information sharing respectively.

In the role identification process, we have decided that the *Surveyor* role has to satisfy the *Find survivors* and *Assess & report damage* subgoals. To satisfy these two subgoals, the subgoals under them which are *Search area*, *Identify survivors*, *Report survivors*, *Make visual documentation* and *Report visual documentation* should be satisfied. The tasks of the role are then specified so as to satisfy these subgoals. A task that performs an exploration using different coverage path planning approaches such as randomised, cellular decomposition, etc. satisfies the *Search area* subgoal. This task requires motion capability and a positioning system such as Global Positioning System (GPS). A task that can identify survivors using an infra red (IR) imaging system can satisfy the *Identify survivors* subgoal. This task requires infra red imaging capability and a positioning capability to identify the location of survivors. A task that performs video streaming or picture archiving satisfies the *Make visual documentation* subgoal. This task requires a video camera. The *Report Survivors* and *Report visual documentation* subgoals can be satisfied by a task that sends a periodic or event triggered report to the manager role of the role containing this task. Using the same argument for the remaining roles, the tasks associated with each role and the corresponding required capabilities are identified as shown in Table 6.1. This serves as a basis for role specification as shown in Section 6.5.

Now that we have identified the roles in our system, we can outline the interaction among them by extracting possible events that might occur in the system from the mission statements as shown in Figure 6.4. The *Commander* role initiates the interaction by providing the map of the disaster area to the *Aggregator* role, which in

Role Type	Tasks	Required Capabilities
Surveyor	Explore, IdentifySurvivor, Report	motion, camera, Infrared imaging, GPS
Aggregator	BuildMap, AssessRisk	motion, powerful processing
Transporter	Transport	motion, lifting, GPS
Medic	AssistSurvivor	motion, medical, GPS
Hdetector	DetectHazard	motion, GPS, chemical and biological hazard detection
Relay	RelayFunction	motion, longrange communication
Commander	ManageEarthQuakeDisaster	motion, longrange communication

Table 6.1: Role Types and Associated Tasks

turn notifies this area to the *Hazard-detector* and *Surveyor* roles so that they can look for hazardous materials, search for survivors and assess the damage respectively. When the *Hazard-detector* role detects a hazard, it marks the hazard on the map and reports the update to the *Aggregator*, which uses this information to create a map showing contaminated areas. The *Surveyor* role takes a picture of the environment and sends it to the *Aggregator* while it is searching for a survivor. The *Aggregator* uses this information to perform damage assessment as well as predict potential collapses. When the *Surveyor* role finds a survivor, it informs the location of the survivor to the *Aggregator* role, which assesses the risk this survivor is exposed to, using the contamination map and its potential-collapse prediction, and decides whether this survivor should be rescued at all and if so whether the survivor should be given priority. The *Aggregator* communicates its risk assessment to the *Surveyor*, and if the assessment is to rescue the survivor, the *Surveyor* informs the location of the survivor to all *Transporter*s it has access to. The *Transporter*s will compute their distance to the survivor and reply with that information, which the *Surveyor* uses to select one among them. If the risk assessment indicates the need for emergency medical assistance, the *Surveyor* contacts both *Medic* and *Transporter* roles.

In a similar manner to the role identification process, some of the interactions among roles are dictated by design choices. For example, because we have chosen an auction based approach for selecting a UAV to transport a survivor among a number of UAVs enacting the *Transporter* role, the *Surveyor* role communicates the location of a survivor to all *Transporter*s. The same applies to *Medic* roles. On the other hand, the

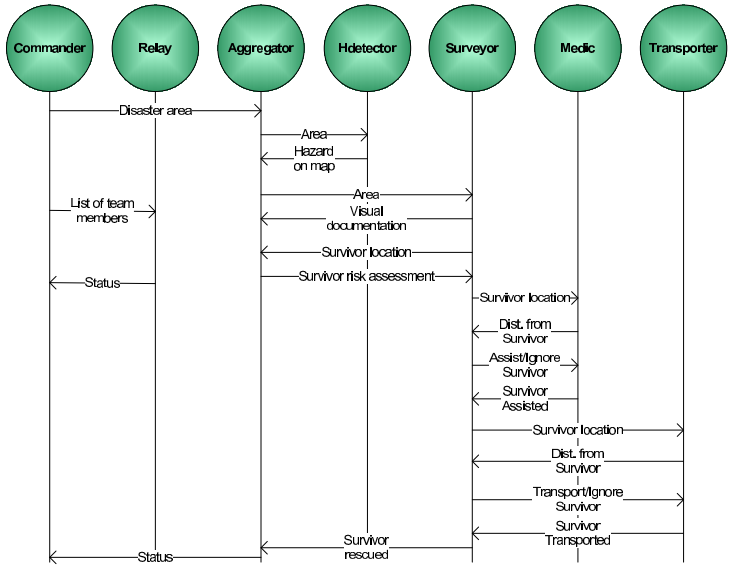


Figure 6.4: Role Interactions

Aggregator role makes the decision of which UAV surveys what area by itself, which can be observed in the interaction diagram where the *Aggregator* informs the area to the *Surveyor* role without any prior negotiation.

The policies of a role can be identified from the goals as well as the mission statement. Some of the events that trigger these policies can be identified from the outline of role interactions discussed in section 6.4. The goals, which themselves are identified from the mission statement can serve as a basis for specifying some fundamental policies such as *if a survivor is injured call Medic*. However, since the goal identification process does not consider operational details, it is necessary to refer to the mission statement in order to identify operational policies such as *give priority to survivors in contaminated area if they have a chance to survive*.

6.5 Role Specification

In the previous section, we have presented a mission refinement process as a basis for the administrator's mission specification task. In this section, we will show how the mission administrator can specify the roles in the search and rescue mission. Recalling the role specification approach presented in Chapter 3, a role specification has the elements – (1) policies relating to the role, (2) the tasks of the role, (3) operations and notification that are provided by the role, (4) operations and notifications

that are required by the role, and (5) the capabilities requirement of the role. The role specification process mainly consists of deciding what tasks to include in the role, determining the capability requirement of the role based on the tasks, deciding the interface to be exposed to collaborating roles based on role interactions and finally specifying policies (role missions) that dictate the role behaviour. For the search and rescue mission, the administrator specifies the *Surveyor*, *Aggregator*, *Transporter*, *Medic*, *Relay*, *Hazard-detector* and *Commander* roles as shown in Figures 6.5 & 6.6, and Figures B.1 - B.7 (Appendix B).

Figures 6.5 & 6.6 show the specification for the *Surveyor* role. This role contains three tasks – $Explore$, $IdentifySurvivor$ and $Report$ – of which one of them, $Explore$, has some of its operations (e.g., $setArea(input = dmrc.util.Area)$, $getPicture(output = dmrc.util.DmrcImage)$) and notifications (e.g., $batteryLevel$) exposed, i.e., become part of the external provided interface of the role. As a result policies of other roles such as the *Commander*, *Aggregator*, etc. can invoke these exposed operations and can be triggered by these exposed notifications. The other two tasks do not have exposed operations and notifications. Consequently, none of the operations from these tasks can be invoked by remote policies and none of the notifications generated by these tasks can trigger remote policies. Local policies, i.e., policies of the *Surveyor* role, however, can invoke operations from these tasks and can be triggered by notifications generated by these tasks (e.g., $survivorDetected$). In order to make the presentation manageable, not all local operations and notifications are shown in the figures. The notification, $survivorRescued$, is an exposed notification of the role that can be used by remote roles. This is generated by the role itself as a result of its management functions or interaction with other roles as opposed to those, such as $survivorDetected$, generated by the tasks in the role. As can be seen from the specification, the *Surveyor* role requires a number of operations and notifications from other roles. All the operations and notifications listed within the $require$ tag are those that are provided to the *Surveyor* role by collaborating roles. The specification for the remaining roles is shown in Appendix B (Figures B.1 - B.7).

```
<xml>
  <role name='Surveyor'>
    <policy uri='http://192.168.0.1/search_rescue/policy/surveyor'/>
    <tasks>
      <task name='Explore'>
        <expose>
          <operations>
            <operation name='setArea'>
              <argument>
                <name>area</name>
                <type>dmrc.util.Area</type>
              </argument>
            </operation>
            <operation name='getPicture'>
              <result>
                <name>picture</name>
                <type>dmrc.util.DmrcImage</type>
              </result>
            </operation>
            <operation name='returnToBase'/>
          </operations>
          <notifications>
            <notification name='batteryLevel'>
              <attribute name='name'/>
              <attribute name='level'/>
            </notification>
          </notifications>
        </expose>
        <local>
          <operations>
            <operation name='distanceFrom'>
              <argument>
                <name>from</name>
                <type>dmrc.util.Location</type>
              </argument>
               <argument>
                <name>to</name>
                <type>dmrc.util.Location</type>
              </argument>
              <result>
                <name>distance</name>
                <type>Double</type>
              </result>
            </operation>
        </local>
      </task>
      <task name='IdentifySurvivor'>
        <local>
         <notifications>
          <notification name='survivorDetected'>
           <attribute name='name'/>
           <attribute name='survivorLocation'/>
          </notification>
         </notifications>
        </local>
      </task>
      <task name='Report'/>
    </tasks>
    <expose>
      <notifications>
        <notification name='survivorRescued'>
          <attribute name='name'/>
          <attribute name='survivorLocation'/>
        </notification>
      </notifications>
    </expose>
    <local>
      <notifications>
        <notification name='UAVFailure'>
          <attribute name='name'/>
          <attribute name='uav'/>
        </notification>
      </notifications>
    </local>
```

Figure 6.5: Search & Rescue Role Specification - Surveyor Role (Part 1)

```
75      <require>
76        <operations>
77          <operation name='uploadVisualDcoument'>
78            <argument>
79              <name>visualDocument</name>
80              <type>dmrc.util.Media</type>
81            </argument>
82          </operation>
83          <operation name='assist'>
84            <argument>
85              <name>survivorLocation</name>
86              <type>dmrc.util.Location</type>
87            </argument>
88          </operation>
89          <operation name='transport'>
90            <argument>
91              <name>survivorLocation</name>
92              <type>dmrc.util.Location</type>
93            </argument>
94            <argument>
95              <name>destination</name>
96              <type>dmrc.util.Location</type>
97            </argument>
98          </operation>
99          <operation name='assessRisk'>
100           <argument>
101             <name>survivorLocation</name>
102             <type>dmrc.util.Location</type>
103           </argument>
104         </operation>
105         <operation name='measureMetric'>
106           <argument>
107             <name>roleId</name>
108             <type>dmrc.util.RoleIdentity</type>
109           </argument>
110           <argument>
111             <name>metricType</name>
112             <type>String</type>
113           </argument>
114           <result>
115             <name>metricValue</name>
116             <type>Double</type>
117           </result>
118         </operation>
119       </operations>
120       <notifications>
121         <notification name='riskAssessed'>
122           <attribute name='name'/>
123           <attribute name='survivorLocation'/>
124           <attribute name='riskLevel'/>
125         </notification>
126         <notification name='survivorAssissted'>
127           <attribute name='name'/>
128           <attribute name='survivorLocation'/>
129         </notification>
130         <notification name='survivorTransported'>
131           <attribute name='name'/>
132           <attribute name='survivorOriginalLocation'/>
133           <attribute name='survivorNewLocation'/>
134         </notification>
135       </notifications>
136     </require>
137     <capability>
138       <require>
139         <type>motion</type>
140         <type>camera</type>
141         <type>IRImaging</type>
142         <type>GPS</type>
143       </require>
144     </capability>
145   </role>
146 </xml>
```

Figure 6.6: Search & Rescue Role Specification - Surveyor Role (Part 2)

6.6 Mission Class Specification

Once the roles of the mission are specified, the next step is organising these roles in a hierarchy that would facilitate the search and rescue goal. This is achieved by the mission class specification, which organises roles into a management hierarchy. For the search and rescue mission, the organisation shown in Figure 6.7 is used, using the roles' interactions as a basis for deciding the management relations between them. The decision, which is made by the mission administrator, may take a number of factors into consideration including:

1. The ability of a role to provide the information necessary for initiating activities, e.g., the *Commander* role provides information about the disaster area and hence should be at the top of the hierarchy.

2. Efficient use of information, e.g., the *Aggregator* role uses information from the *Hazard-detector* role to provide the *Surveyor* with a risk assessment for a survivor and hence the *Aggregator* role should manage the *Hazard-detector* and *Surveyor* roles.

3. Frequency of interaction, e.g., the *Medic* and *Transporter* roles interact frequently, and only, with the *Surveyor* role and hence these roles should be managed by the *Surveyor* role.

Using the management hierarchy and mission parameters, the mission class for the search and rescue mission is specified as shown in Figure 6.8. The mission parameters are specified in lines 6 to 12. The *comTimeout* and *failureTimeout* parameters specify timeouts, in milliseconds, for intermittent communication link and permanent communication link or UAV failure detection. The *discoveryRate*, *optimisationRate* and *stateUpdateRate* specify the frequency of discovery beacon broadcast, optimisation, and state update message sending respectively.

The *minBatteryLevel* parameter specifies the minimum battery power, in milliampere hour, a UAV is required to have to commence or continue its participation in the mission. The *disasterArea* parameter specifies the area where the search and rescue mission is performed.

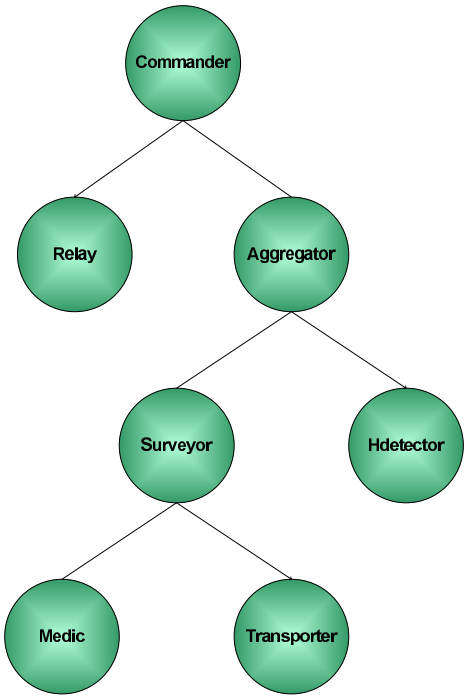


Figure 6.7: Management Hierarchy

The rest of the mission specification specifies the remaining roles participating in the mission with their maximum cardinality and encodes the management relationship among them. For example, in lines 14 - 20, we can observe that the *Commander* is responsible for managing the *Relay* and *Aggregator* roles. However, as the mission class is a skeletal form of the mission, neither the actual number of roles nor their behaviour is yet specified. This delayed cardinality and behaviour specification enables us to use this mission class for other search and rescue missions.

After specifying the mission class, the administrator specifies a mission class instantiation that is tailored to meet the context of the search and rescue mission at residential complex Alpha (the disaster area as stated in Section 6.2). The administrator decides on the values of mission parameters and role cardinalities. The role behaviours, which are specified using policies, are provided through URIs that are used by the UAV assigned to the role (or the *Commander* role if it pre-fetches the policies before starting the team formation) to download the policies from the policy repository.

The mission parameters such as failure timeouts and minimum battery level are specified taking the size and nature of the mission area into consideration. The maximum distance a UAV might travel from its managing UAV depends on the size

```
<xml>
<missionClassParameters>
    <Name>SearchAndRescue</Name>
</missionClassParameters>
<missionParameters>
    <comTimeout>int</comTimeout>
    <failureTimeout>int</failureTimeout>
    <discoveryRate>int</discoveryRate>
    <optimisationRate>int</optimisationRate>
    <stateUpdateRate>int</stateUpdateRate>
    <minBatteryLevel>double</minBatteryLevel>
    <disasterArea>URI</disasterArea>
</missionParameters>
<commanderBehaviour>
    <cardinality>int</cardinality>
    <roleManagement>
        <manages>relay</manages>
        <manages>aggregator</manages>
    </roleManagement>
</commanderBehaviour>
<aggregatorBehaviour>
    <cardinality>int</cardinality>
    <roleManagement>
        <manages>surveyor</manages>
        <manages>hdetector</manages>
    </roleManagement>
</aggregatorBehaviour>
<surveyorBehaviour>
    <cardinality>int</cardinality>
    <roleManagement>
        <manages>medic</manages>
        <manages>transporter</manages>
    </roleManagement>
</surveyorBehaviour>
<hdetectorBehaviour>
    <cardinality>int</cardinality>
    <roleManagement/>
</hdetectorBehaviour>
<relayBehaviour>
    <cardinality>int</cardinality>
    <roleManagement/>
</relayBehaviour>
<medicBehaviour>
    <cardinality>int</cardinality>
    <roleManagement/>
</medicBehaviour>
<transporterBehaviour>
    <cardinality>int</cardinality>
    <roleManagement/>
</transporterBehaviour>
</xml>
```

Figure 6.8: Search & Rescue Mission Class Specification

of the mission area and the materials in the mission area. This distance helps in providing a relative estimate of the duration to wait before deciding that either a communication link or a UAV has permanently failed, i.e., communication and failure timeouts. For example, if the buildings are made of steel structure, frequent radio signal blockage might occur when a UAV is inside the remnants of the buildings.

```
<xml>
<missionParameters>
    <comTimeout>3000</comTimeout>
    <failureTimeout>7000</failureTimeout>
    <discoveryRate>500</discoveryRate>
    <optimisationRate>20000</optimisationRate>
    <stateUpdateRate>400</stateUpdateRate>
    <minBatteryLevel>1000</minBatteryLevel>
    <disasterArea>http://192.168.0.1/map/alpha</disasterArea>
</missionParameters>
<commander>
    <cardinality>1</cardinality>
    <policy>http://192.168.0.1/policy/commander</policy>
</commander>
<aggregator>
  <cardinality>1</cardinality>
    <policy>http://192.168.0.1/policy/aggregator</policy>
</aggregator>
<surveyor>
    <cardinality>1</cardinality>
    <policy>http://192.168.0.1/policy/surveyor</policy>
</surveyor>
<hdetector>
    <cardinality>1</cardinality>
    <policy>http://192.168.0.1/policy/hdetector</policy>
</hdetector>
<relay>
    <cardinality>1</cardinality>
    <policy>http://192.168.0.1/policy/relay</policy>
</relay>
<medic>
    <cardinality>3</cardinality>
    <policy>http://192.168.0.1/policy/medic</policy>
</medic>
<transporter>
    <cardinality>3</cardinality>
    <policy>http://192.168.0.1/policy/transporter</policy>
</transporter>
</xml>
```

Figure 6.9: Search & Rescue Mission-Class Instance Specification – for Mission Area Alpha

Other factors may also indirectly contribute to the choice of values for the failure timeout parameters. For example, if an aftershock is expected in a short time, a more fast-paced rescue mission may be necessary so the failure timeouts need to be shorter and battery levels need to be higher. The value of the mission parameters can be changed at runtime using policies.

The mission class can now be instantiated using the mission class instantiation specification shown in Figure 6.9 resulting in the team shown in Figure 6.10.

The mission class specification used for residential complex Alpha can be reused for another mission with another mission-class instance specification that has larger cardinalities, as shown in Figure 6.11, for a search and rescue mission in residential complex Beta, which has a larger area than Alpha.

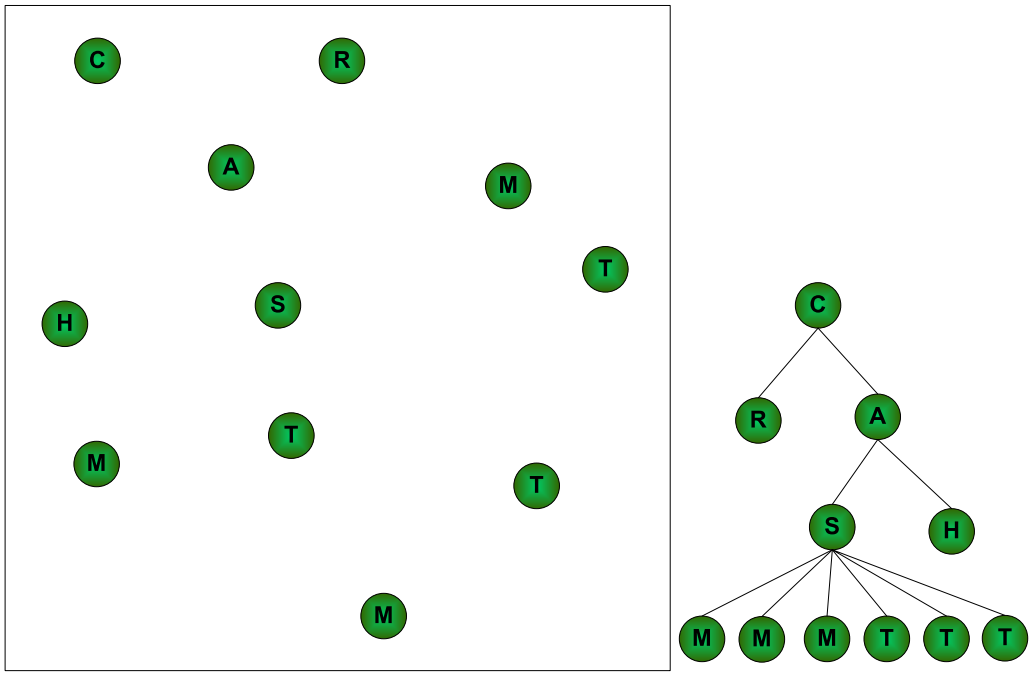


Figure 6.10: UAV Team for Search & Rescue Mission of Residential Complex Alpha

The cardinality of a role type is the total number of role instances of that type allowed to be assigned by a single manager role. The total number of role instances of a given role type is determined by the cardinality of that role and the cardinality of its manager role. For example, assuming that $cardinality(C) = 1$, the total number of *Surveyor* roles for our mission class is $cardinality(A)cardinality(S)$, which gives us one *Surveyor* role for mission instance Alpha and two *Surveyor* roles for mission instance Beta. The total number of roles in the mission can be computed as:

$(cardinality(C) + cardinality(C)\ (cardinality(R) + cardinality(A)\ (cardinality(S)$
$(cardinality(M) + cardinality(T)) + cardinality(H))))$

In addition to the role cardinalities, the values for the mission parameters are also adapted to suite the mission. The *Aggregator*s can partition the area and allocate the *Surveyor*s to different partitions as shown in Figure 6.12.

```
<missionParameters>
    <comTimeout>3000</comTimeout>
    <failureTimeout>7000</failureTimeout>
    <discoveryRate>500</discoveryRate>
    <optimisationRate>10000</optimisationRate>
    <stateUpdateRate>400</stateUpdateRate>
    <minBatteryLevel>2000</minBatteryLevel>
    <disasterArea>http://192.168.0.1/map/beta</disasterArea>
</missionParameters>
<commander>
    <cardinality>1</cardinality>
    <policy>http://192.168.0.1/policy/commander</policy>
</commander>
<aggregator>
    <cardinality>2</cardinality>
    <policy>http://192.168.0.1/policy/aggregator</policy>
</aggregator>
<surveyor>
    <cardinality>2</cardinality>
    <policy>http://192.168.0.1/policy/surveyor</policy>
</surveyor>
<hdetector>
    <cardinality>1</cardinality>
    <policy>http://192.168.0.1/policy/hdetector</policy>
</hdetector>
<relay>
    <cardinality>2</cardinality>
    <policy>http://192.168.0.1/policy/relay</policy>
</relay>
<medic>
    <cardinality>1</cardinality>
    <policy>http://192.168.0.1/policy/medic</policy>
</medic>
<transporter>
    <cardinality>1</cardinality>
    <policy>http://192.168.0.1/policy/transporter</policy>
</transporter>
</xml>
</xml>
```

Figure 6.11: Search & Rescue Mission Class Instance Specification (for Mission Area Beta) - Role Cardinalities & Behaviours

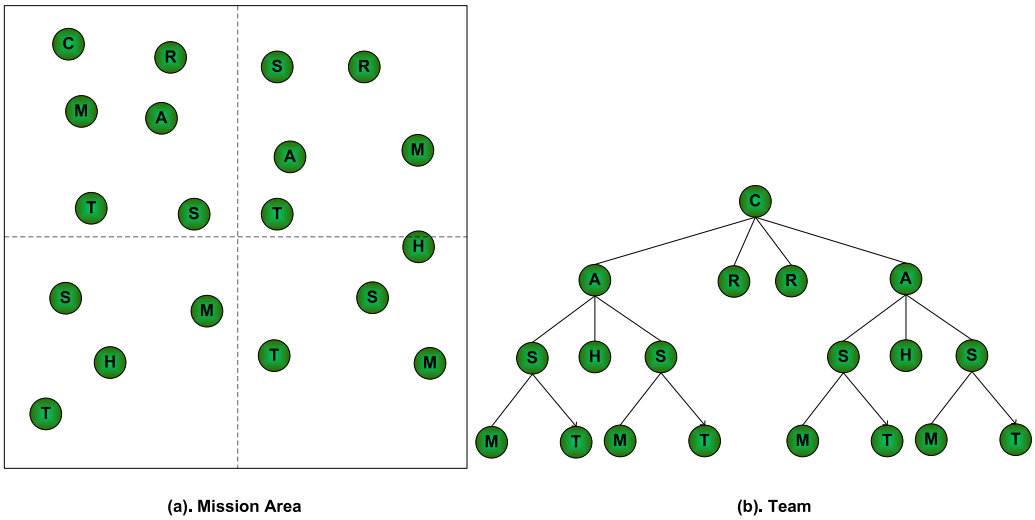


Figure 6.12: UAV Team for Search & Rescue Mission of Residential Complex Beta

6.7 Policy Specification

In the previous sections, we have specified the roles for the search and rescue mission that serve as building blocks for the team. We have also specified the mission class, which defines the team structure and the mission instance, which defines the actual form of the team in accordance with the team structure. The individual behaviours of roles, which are the building blocks of the team as well as the team as a whole, are specified by policies as shown in the next two sections where a number of policies that specify adaptive team formation and mission behaviours are presented.

The search and rescue mission, as can be observed from the mission statement (Section 6.3) and the mission specification (Section 6.4), has a number of conditional actions and interactions that depend on the context of the mission. Policies can capture these behaviours thereby allowing an adaptive search and rescue mission.

6.8 Search and Rescue Team Formation

The search and rescue team is formed in accordance with the mission specification as a result of interacting UAVs that are loaded with the self-management framework. As mentioned in Chapter 3, all elements of the framework are stored in a domain structure. Consequently, loading the UAV with the self-management framework involves creating these elements and putting them in the domain structure. The bootstrap process used to load the common elements of the framework on all UAVs is discussed in Chapter 7. Other elements such as roles are added later on through policies as part of the mission execution. Note that the dynamic team formation involves UAVs that are already loaded with the management framework.

The search and rescue team formation is initiated manually by the mission administrator through loading the mission class specification with the required mission instance specification onto the *Commander* role. The assignment of the *Commander* role, unlike other roles, is done by the administrator. The assignment is performed by loading a mission startup policy, shown in Figure 6.13, to a UAV that is running the self-management framework. The administrator then triggers this policy to startup the mission. The mission class and instance names as well as the path to

```
1  //Mission startup policy
2  policy := root/factory/ecapolicy create.
3  policy event: /event/missionStartup;
4  action: [ :missionClassSpec :missionInstanceSpec :repository|
5  //Load the Commander role code if it is not already
6  //loaded on to the domain structure
7  root/factory at: "commander" ifAbsent: [
8  root/factory at:"commander"
9          put: (root load: "dmrc.role.Commander"). ].
10 //Create an entry for the Commander role in the
11 //domain structure if it is not already there
12 root/role at: "commander" ifAbsent: [
13 root/role at: "commander" put: ((root load: "Domain") create). ].
14 //Create a Commander role instance
15 root/role/commander at: "commander"
16         put: (root/factory/commander create).
17 //Load the mission class and instance
18 //specification, i.e., startup the mission
19 root/role/commander/commander
20 mcspec:"http://"+repository+"/mission_class_spec_case_study_dist_role"
21 instspec:"http://"+repository+"/mission_class_instance_spec_case_study_dist_role".
22 ].
23 policy active: true.
```

Figure 6.13: Search & Rescue Mission Startup

the repository are passed to the action part of the policy through the *missionStartup* event attributes.

In addition to the mission startup approach shown here, which uses a policy local to the *Commander* role, the administrator also has a choice of loading the mission class and instance specifications remotely (i.e., by importing the *Commander* role reference to its management console and invoking the $mcspec : instspec :$ operation on the role).

Upon mission startup, the *Commander* role loads policies associated to it, shown in Figures 6.15 & 6.16, including role assignment policies, which drive the team formation, and starts the discovery service. Figure 6.14 illustrates the dynamics of the framework elements as dictated by the *Commander*'s policies. The solid labeled lines with arrows indicate policies (e.g., $newUAV\ [notAllAssigned]\ addUav$ stands for the $addUav$ policy, with the condition $notAllAssigned$, which is triggered by the $newUAV$ event). The arrow indicates the (target) framework element that is managed by the policy. The numbered broken lines with arrows indicate the interactions initiated by the policies.

The first policy, $startDiscovery$, shown in Figure 6.15 (lines 2 - 21) creates the discovery service, binds it to the role that is going to perform the discovery and assignment (in this case the *Commander* role) and activates the *Relay* and *Aggregator* role assignment policies. It also sets (1) the threshold factor (σ), which is used by the com-

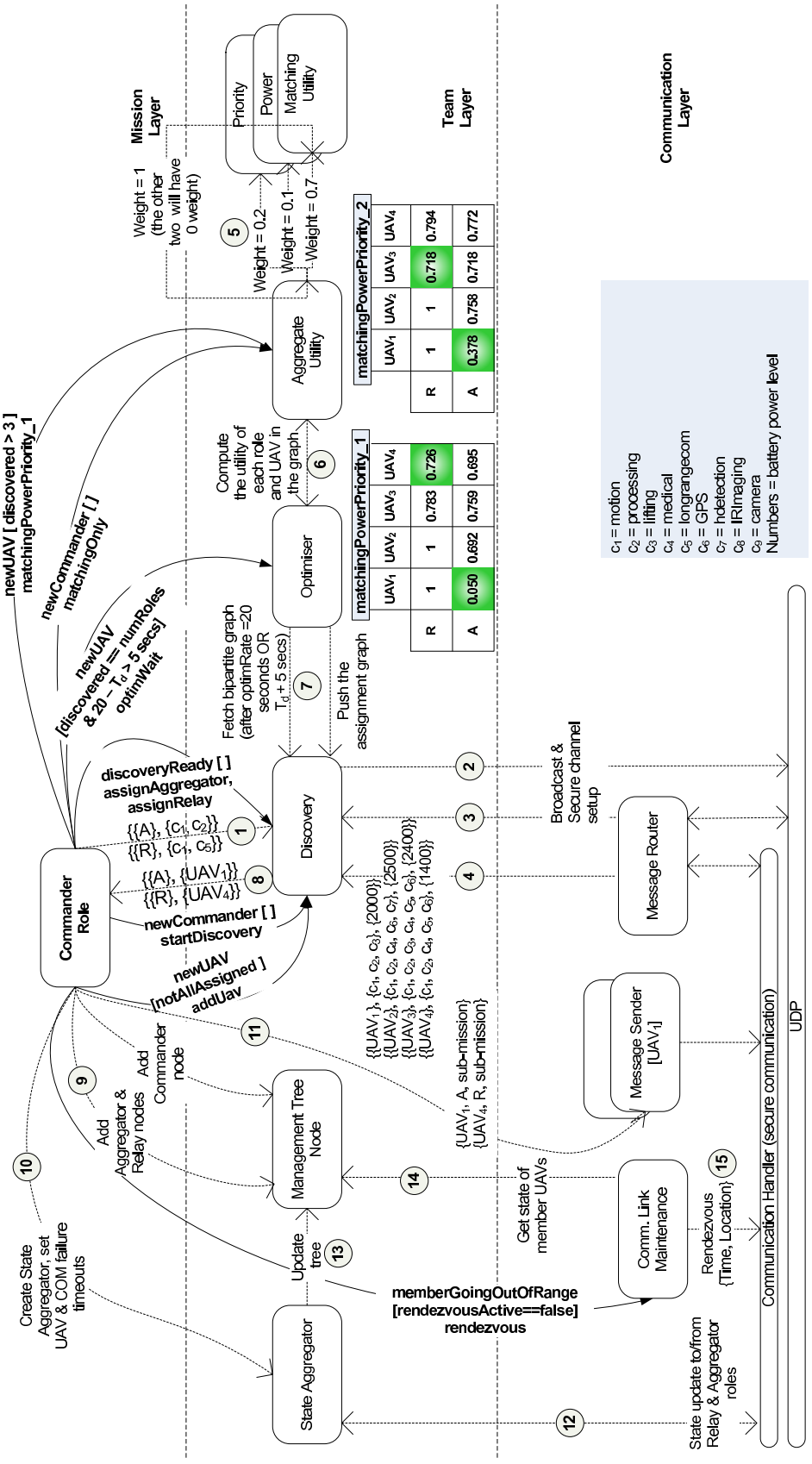


Figure 6.14: Instance of the Management Framework on the Commander UAV

munication link maintenance element to determine the range threshold as discussed in Chapter 5, and (2) the leader role for movement-adaptation based communication link maintenance (as discussed in Chapter 5). The policy also turns off the discovery service's trigger mechanism that makes the optimiser start optimisation when as many UAVs are discovered as are required to fulfil available roles. This behaviour, which may speed up the team formation when there are sufficient numbers of UAVs, could result in a long or indefinite wait when sufficient number of UAVs are not discovered in a short period of time. When this behaviour is disabled, the optimiser waits for a specified period of time (set by a policy and/or mission parameter) before starting optimisation thereby either allowing for the possibility of more UAVs to be discovered or cutting short the waiting period (and assigning some of the roles using the available UAVs) in case insufficient UAVs to fulfil available roles, are discovered within the prescribed time. The policy's last operation instructs the discovery service to generate a $discoveryReady$ notification, which as we later on will see, triggers other policies, and starts broadcasting. The immediate interactions initiated by this policy are the broadcast and secure channel setup (if one or more UAVs reply to the broadcast) as shown in Figure 6.14 (numbers 2 and 3 respectively).

The second policy, $newUAV$, shown in Figure 6.15 (lines 24 - 29) adds newly discovered UAVs to the bipartite graph built by the discovery service using available roles and UAVs that can be potential team members. The interactions initiated by this policy are those that involve the request and reply of low-level capability descriptions (4) including battery power levels encoded in XML. In the diagram (Figure 6.14), the replies of four UAVs are shown in a flattened format.

The policies shown in Figure 6.15 (lines 31 - 42) are assignment policies for the *Aggregator* and *Relay* role. These policies effectively set the low-level capability requirements of the roles as shown in Figure 6.14. The *Aggregator* assignment policy also sets the priority of the *Aggregator* role (line 33), which is used for computing the utility of this role (if the priority utility is considered). The higher the number the higher the utility. In our prototype implementation, unless the priority level is explicitly set by a policy, the default levels are 5. The last policy (lines 45 - 55) is used to import role references of managed roles (*Relay* and *Aggregator*) and configure the state update sent by them. A similar policy (used by the *Aggregator* role) is discussed in detail later.

```
1  //Discovery startup policy
2  policy := root/factory/ecapolicy create.
3  policy event: /event/newCommander;
4  action: [ :name :role :instance|
5  //Create the discovery and optimiser services.
6  //The code for these services should already be
7  //loaded on to the UAV with other framework elements.
8  root at: "discovery" put: (root/factory/discovery create).
9  root at: "optimiser" put: (root/factory/optimiser create).
10 root/discovery setRole: role instance: instance.
11 //Enable the role assignment policies
12 root/policy/assignAggregatorOpt active: true.
13 root/policy/assignRelayOpt active:true.
14 root/comlinkmaintainer thresholdFactor: "0.8".
15 root/comlinkmaintainer setLeader: "surveyor"
16 //Turn off the discovery trigger for optimisation
17 root/discovery/opportunisticOpt : false.
18 //Generate the discoveryReady notification and start broadcasting.
19 root/discovery ready.].
20 root/policy at: "startDiscovery" put: policy.
21 policy active: true.
22 //If all of the roles managed by the Commander are not assigned
23 //add this UAV as a potential member
24 policy := root/factory/ecapolicy create.
25 policy event: /event/newUAV;
26 condition:[root/discovery notAllAssigned.];
27 action: [ :lowLevelCap :uav |root/discovery adduav: uav cap: lowLevelCap].
28 root/policy at: "addUav" put: policy.
29 policy active: true.
30 //Aggregator assignment policy
31 policy := root/factory/ecapolicy create.
32 policy event: /event/discoveryReady;
33 action: [root/utility/priority setpriority: 4 role: "aggregator".
34 root/discovery has:#("motion" "processing") assign:"aggregator"].
35 root/policy at: "assignAggregatorOpt" put: policy.
36 policy active: false.
37 //Relay assignment policy
38 policy := root/factory/ecapolicy create.
39 policy event: /event/discoveryReady;
40 action: [root/discovery has:#("motion" "longrangecom") assign:"relay"].
41 root/policy at: "assignRelayOpt" put: policy.
42 policy active: false.
43 //Import references of managed roles (Surveyor and Hazard–detector),
44 //set state update type
45 policy := root/factory/ecapolicy create.
46 domainFactory := root load: "Domain".
47 policy event: /event/roleAssigned;
48 action: [ :name :role :instance :address :path|
49 root/role at: role ifAbsent:[root/role at:role
50     put:(domainFactory create).].
51 (root/role resolve: role) at: instance
52 put:(root import: path from: address).
53 (root/role resolve: (role+"/"+instance)) setStateUpdateType: "management".].
54 root/policy at: "importRole" put: policy.
55 policy active: true.
```

Figure 6.15: Search & Rescue Mission – Commander Role, Assignment & Discovery Policies

In the next set of policies of the *Commander* shown in Figure 6.16, the first policy, *matchingOnly* (lines 4 - 8) sets the weight of the matching utility to 1 thereby forcing the utility aggregation process to ignore all the other utilities. Since this policy is triggered by the *newCommander* event the *Commander* role, as soon as it starts,

will have the behaviour of selecting UAVs for the *Relay* and *Aggregator* roles based on the UAVs' capabilities only (i.e., whether the capabilities of the UAVs match the requirements of the roles). However, if the number of discovered UAVs exceeds three, the second policy (lines 11 - 19), $matchingPowerPriority_1$, which is triggered by the $newUAV$ event resets the weight parameters enabling the consideration of other utilities (role priorities and battery power level) in computing the aggregate utility. Since the availability of UAVs is not known a priori, this adaptive behaviour is useful in order to relax or tighten the vetting process with respect to the quality of UAVs depending on availability of UAVs. The interactions initiated by these policies are those that involve the *Optimiser*, *Aggregate Utility*, *Priority*, *Matching* and *Power* elements (5, 6) as shown in Figure 6.14.

The $optimRate$ policy (lines 33 - 40) adapts the waiting period before optimisation, which is measured from the time the first UAV is discovered up to the time optimisation starts, depending on the number of discovered UAVs. This parameter, i.e., the waiting period, was initially specified to be 20 seconds in the mission instance specification (Figure 6.9). The policy will shorten this period if the required number of UAVs to fulfil roles are discovered in a shorter period of time. If the number of discovered UAVs, during the period T_d, fulfils the number of roles waiting to be assigned and the remaining waiting period $(20 - T_d)$ at this time is longer than 5 seconds, the policy results in only 5 more seconds of waiting to see if more UAVs would be discovered before starting the optimisation as opposed to $20 - T_d$ seconds.

The last policy, $rendezvous$ (lines 42 - 47), initiates the rendezvous algorithm when a $memberGoingOutOfRange$ notification is received. As discussed in Chapter 5, the rendezvous-based approach is managed by the *Commander* and consequently the policy that manages the rendezvous-based communication link maintenance is specified only on the *Commander* role. The *Commander* role monitors the distance among member UAVs using the state update messages and when it detects a trend that would result in potential communication link disconnection, it generates the $memberGoingOutOfRange$ notification, which will trigger the policy that sets up the rendezvous if one is not already setup. The immediate interactions initiated by these policies involve those between the *Optimiser* and the *Discovery* service (7), and the *Communication Link Maintenance*, *Management Tree Node*, *Communication Handler* elements (13 & 14) as shown in Figure 6.14.

```
1  //Set the matching utility weight.
2  //Setting the weight of one of the utilities
3  //to one will set the other weights to zero.
4  policy := root/factory/ecapolicy create.
5  policy event: /event/newCommander;
6  action: [root/utility/aggregate utility:"matching" setweight: "1".].
7  root/policy at: "matchingOnly" put: policy.
8  policy active: true.
9  //If the number of discovered UAVs is greater
10 //than three, set the weights for the three utilities.
11 policy := root/factory/ecapolicy create.
12 policy event: /event/newUAV;
13 condition:[root/discovery discovered > 3];
14 action: [
15 root/utility/aggregate utility:"matching" setweight: "0.7".
16 root/utility/aggregate utility:"power" setweight: "0.1".
17 root/utility/aggregate utility:"priority" setweight: "0.2".].
18 root/policy at: "matchingPowerPriority_1" put: policy.
19 policy active: true.
20 //Different weights
21 policy := root/factory/ecapolicy create.
22 policy event: /event/newUAV;
23 condition:[root/discovery discovered > 3
24 action: [
25 root/utility/aggregate utility:"matching" setweight: "0.5".
26 root/utility/aggregate utility:"power" se "0.3".
27 root/utility/aggregate utility:"priority" setweight: "0.2".].
28 root/policy at: "matchingPowerPriority_2" put: policy.
29 policy active: false.
30 //Set the waiting period before optimisation (in miliseconds)
31 //if the number of discovered UAVs is equal to
32 //the number of roles waiting to be assigned.
33 policy := root/factory/ecapolicy create.
34 policy event: /event/newUAV;
35 condition:[(root/discovery discovered) ==
36     (root/role/commander/commander numRoles)];
37 action: [(root/optimiser remainingperiod > 5000)
38 ifTrue: [root/optimiser setoptimrate: 5000]].
39 root/policy at: "optimRate" put: policy.
40 policy active: true.
41 //Initiate the rendezvous algorithm
42 policy := root/factory/ecapolicy create.
43 policy event: /event/memberGoingOutOfRange;
44 condition:[(root/comlinkmaintainer rendezvousActive)==false];
45 action: [root/comlinkmaintainer startRendezvous.].
46 root/policy at: "rendezvous" put: policy.
47 policy active: true.
```

Figure 6.16: Search & Rescue Mission – Commander Role Optimisation & Communication Policies

Figure 6.14 also illustrates the role assignment process using the *Commander* role's policies discussed so far and four UAVs ($UAV_1, UAV_2, UAV_3, UAV_4$) with the capabilities shown in the figure. The *Aggregator* and *Relay* roles are assigned to UAV_1 and UAV_4 respectively (the optimiser's decision is highlighted in the role-uav matrix shown in the figure) as these UAVs have the lowest cost (highest aggregate utility) for the corresponding roles. The values of the aggregate utility depend on the weight given to each utility type and the utility functions. The utility functions included in

the prototype implementation and used for the case study are shown below.

$$utility(uav) = \begin{cases} 0 & level(uav) \leq minBatteryLevel \\ \frac{level(uav) - minBatteryLevel}{level(uav)} & level(uav) > minBatteryLevel \end{cases}$$

$$utility(role) = \begin{cases} \frac{priority}{priorityLevels} \end{cases}$$

$$utility(uav, role) = \begin{cases} \frac{k}{(prov_{cap}(uav) \ \Delta \ req_{cap}(role))penalty+1} & prov_{cap}(uav) \ \Delta \ req_{cap}(role) \leq k \\ \frac{prov_{cap}(uav) \ \Delta \ req_{cap}(role)}{(prov_{cap}(uav) \ \Delta \ req_{cap}(role))penalty+1} & prov_{cap}(uav) \ \Delta \ req_{cap}(role) > k \end{cases}$$

The first function, $utility(uav)$, is a system utility that maps the battery level of UAV uav into a number between 0 and 1. The value of this function depends on the battery power level of the UAV received in the capability summary and the minimum battery level mission parameter set by the mission specification. The second function, $utility(role)$, is a role utility that maps the priority given to a role $role$ into a number between 0 and 1. The value of this function depends on the priority given to the role (through a policy) and the levels of priority defined by the mission (with a default value of 5 levels). The third function, $utility(uav, role)$ is a role-system utility that maps the capability of UAV uav with respect to satisfying the requirement of the role $role$ into a number between 0 and 1. The value of this function depends on the capability of the UAV ($prov_{cap}$), the capability requirement of the role (req_{cap}), the $penalty$ parameter and the Bloom filter parameter k. The $penalty$ parameter is used to amplify the difference in capability for situations where small differences matter. The default value of this parameter is 1 and it can be set through a policy. When the value of this parameter is high, UAVs with only a slightly higher number of capabilities than required by a role will have smaller utilities. The parameter k represents the number of hash functions used by the Bloom filter. As mentioned in Chapter 4, Bloom filters are used to represent the capabilities of UAVs as well as the requirements of roles. A Bloom filter is a space-efficient randomised data structure that can represent a set with n elements by an array of m bits (initially all set to 0) using k independent hash functions (h) that map each item in the universe to a random number uniform over the range 1 to m. For each element c, the bits $h_i(c)$ of the array are set to 1 for $1 \leq i \leq k$. Whether or not c is a member of the set can then be decided by checking whether all $h_i(c)$ are set to 1. The capability of a UAV and the requirement of a role are represented in Bloom filters of identical parameters and the symmetric difference (Δ) set operation is used to compute the difference in capabilities. The

Commander computes the aggregate utility for each discovered UAV using the weights set by the policy $matchingPowerPriority\_1$ (Figure 6.16, lines 11 - 19) and the utility functions. As can be seen in the figure the $matchingPowerPriority\_1$ policy gives much more weight to matching (0.7) than to UAV battery power (0.1) and role priority (0.2). The assignment decision based on this policy is shown in Figure 6.14 where the *Aggregator* is assigned to UAV_1 and the *Relay* is assigned to UAV_4. To illustrate the impact of the choice of weight another weight policy, $matchingPowerPriority\_2$ is included in Figure 6.16 (lines 21 - 29). This policy gives a weight of 0.5 to matching, 0.3 to UAV battery power and 0.2 to role priority. When this policy is activated, the assignment decision changes as can be seen in Figure 6.14 where the *Aggregator* is still assigned to UAV_1 but the *Relay* is assigned to UAV_3, which has much more capabilities than needed by the *Relay* but also has more battery power than UAV_4. In this example, we have changed the weight policies while keeping the number and type of UAVs constant. Given a certain weight policy, if the number and/or types of UAVs are changed and re-optimisation is performed the assignment decision changes in a similar manner and the team becomes destabilised. For this reason, as mentioned before, we do not consider re-optimisation on already assigned roles during failure.

Once the decision as to which UAVs to use for the two roles is made, the *Commander* role creates *Message Sender* elements for UAV_1 and UAV_2, which are used for communications such as role (re)assignment messages that need a reliable channel. It then sends role assignment messages (which contain the sub-mission related to each role) to the UAVs (11) and adds these two roles in its instance of the management tree (9). It also creates a *State Aggregator* element (10), with the node (UAV) and communication link failure timeouts using the values provided through the mission instance specification, which will receive state updates from these two roles (12) and updates the management tree instance (13) accordingly, and generates failure notifications when updates are not received within the given timeouts.

When UAV_1 and UAV_2, which are running the same management framework as the *Commander* UAV (since all UAVs run the same framework), receive the role assignment messages, the *Message Router* elements of these UAVs route the messages to the *Role Manager* elements as shown in Figure 6.18 for the *Aggregator* UAV (UAV_1). This results in the roles being loaded onto the UAVs and their associated policies taking control. The policies related to the *Aggregator* role are shown in Figure 6.17

and the interactions (numbered from 1 - 18) initiated by these policies are shown in Figure 6.18

The $startDiscovery$ policy of the *Aggregator* role also activates the role assignment policies, creates the tasks of the *Aggregator* role (i.e., $BuildMap$ and $AssessRisk$ tasks), configures and binds them to the role. The use of policies for adaptive task creation, configuration and binding has been discussed in detail in Chapter 3. The next four policies, $addUav$, $assignSurveyorOpt$, $assignHdetectorOpt$ and $matchingOnly$ (lines 30 - 49) are also similar to the *Commander* role policies.

In a similar manner to the $startDiscovery$ policy of the *Commander* role, the first policy starts the discovery service that is used by the *Aggregator* role to discover and assign the *Surveyor* and *Hazard-detector* roles. In addition, for the movement-adaptation based communication link maintenance, this policy sets the leader for the *Aggregator* role to be the *Surveyor* (line 9). Unlike the rendezvous approach, which is managed by the *Commander*, the movement-adaptation based approach is distributed and hence each role needs to specify the leader role. As can be seen in the corresponding policies of the roles, in the search and rescue mission, all the roles set the *Surveyor* as their leader which results in each UAV trying to stay within the communication range of either the *Surveyor* UAV or another member UAV that is closer to both itself and the *Surveyor*. If a UAV's movement trend is such that it will not be able to stay within the communication range of the leader UAV (either by staying close to it or by staying close to one that is close to the leader), the *Commander* would be able to detect this trend since it monitors the distance between member UAVs periodically by means of the state update messages. The detection results in the $memberGoingOutOfRange$ event, which triggers the *Commander*'s policy that sets up a rendezvous.

The $importRole$ policy (lines 52 - 60), which is triggered by the $roleAssigned$ event, imports a reference to the *Surveyor* and *Hazard-detector* roles and stores it in the domain structure so that the policies of the *Aggregator* will be able to invoke operations provided by these roles. For example, the policy itself uses the reference to invoke the $setStateUpdateType$ operation on the role corresponding to the imported reference (line 59). These operations set the state update type to *management* which enforces the corresponding role to send a specific state update type. The *Aggregator* role itself sends state updates to its manager (*Commander*) and managed (*Surveyor* and *Hazard-detector*) roles and hence could set its state update type or rely on the default

```
//Start the discovery and optimisation services,
//create, configure and bind tasks of the Aggregator role.
policy := root/factory/ecapolicy create.
policy event: /event/newAggregator;
action: [ :name :role :instance|
root at: "discovery" put: (root/factory/discovery create).
root at: "optimiser" put: (root/factory/optimiser create).
root/discovery setRole:role instance: instance.
root/comlinkmaintainer setLeader: "surveyor".
//Create tasks of the Aggregator role if they are not already created.
((root/task asHash) has: "buildmap") ifFalse:[
root/task at: "buildmap"
           put:((root load:"dmrc.task.BuildMap")create)].
((root/task asHash) has: "assessrisk") ifFalse: [
 root/task at: "assessrisk"
           put:((root load: "dmrc.task.AssessRisk") create)].
//Configure tasks.
root/task/assessrisk bindMotionTask:(/root/task/motion).
root/task/assessrisk bindBuildMapTask: (/root/task/buildmap).
//Bind tasks.
(root/role resolve: (role+"/"+instance))
bindAssessRiskTask: (root/task/assessrisk).
(root/role resolve: (role+"/"+instance))
bindBuildMapTask: (root/task/buildmap).
root/policy/assignSurveyorOpt active: true.
root/policy/assignHdetectorOpt active:true.
root/discovery ready.].
root/policy at: "startDiscovery" put: policy. policy active: true.
//Add discovered UAV as a potential member.
policy := root/factory/ecapolicy create.
policy event: /event/newUAV;
condition:[root/discovery notAllAssigned.];
action: [ :lowLevelCap :uav |root/discovery adduav:uav cap:lowLevelCap].
root/policy at: "addUav" put: policy. policy active: true.
//Surveyor and Hazard-detector role assignment policies.
policy := root/factory/ecapolicy create.
policy event: /event/discoveryReady;
action: [ :name|root/discovery has:#("motion"
"camera" "IRImaging" "GPS")  assign:"surveyor"].
root/policy at: "assignSurveyorOpt" put: policy.
policy active: false.
policy := root/factory/ecapolicy create.
policy event: /event/discoveryReady;
action: [root/discovery has:#("motion" "hdetection" "GPS")  assign:"hdetector"].
root/policy at: "assignHdetectorOpt" put: policy. policy active: false.
policy := root/factory/ecapolicy create.
policy event: /event/newAggregator;
action: [root/utility/aggregate utility:"matching" setweight:1].
root/policy at: "matchingOnly" put: policy. policy active: true.
//Import references of managed roles (Surveyor and Hazard-detector),
//set state update type
policy := root/factory/ecapolicy create.
domainFactory := root load: "Domain".
policy event: /event/roleAssigned;
action: [ :name :role :instance :address :path|
root/role at: role ifAbsent:[root/role at:role put:(domainFactory create).].
(root/role resolve: role) at: instance
put:(root import: path from: address).
(root/role resolve: (role+"/"+instance))  setStateUpdateType: "management".].
root/policy at: "importRole" put: policy. policy active: true.
//Make the Surveyor return to base when its battery power is low
policy := root/factory/ecapolicy create.
policy event: /event/batteryLevel;
condition:[:role :level|(role=="surveyor") &
(level < (root/role resolve: (role+"/"+instance)
 missionParam : "minBatteryLevel")];
action: [ :name :level :role :instance|
    (root/role resolve: (role+"/"+instance)) returnToBase. ].
root/policy at: "surveyorBatteryLow" put: policy. policy active: true.
```

Figure 6.17: Search & Rescue Mission – Aggregator Role Policies

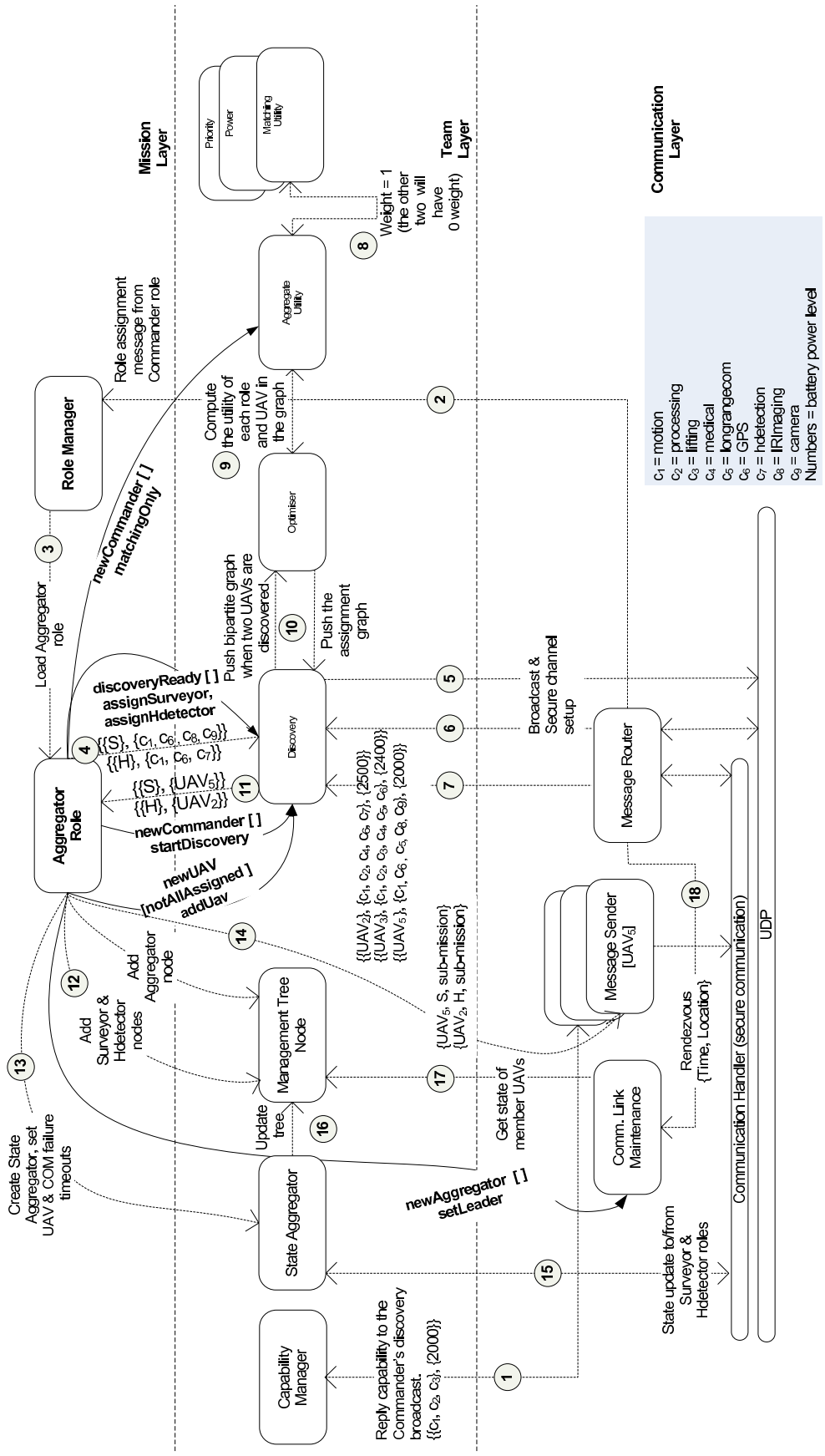


Figure 6.18: Instance of the Management Framework on the Aggregator UAV

setting (which is the case here). By default, all roles send a state update referred to as *alive*, which contains minimal information needed for the functioning of the management framework. The types of state updates are discussed in detail in Chapter 7. The imported role references are used by other policies as well. For example, the $surveyorBatteryLow$ policy (lines 62 - 69), which is triggered by the $batteryLevel$ event, invokes the $returnToBase$ operation on the source role of this event if the battery level is less than the minimum battery level prescribed by the mission specification and the role type is a *Surveyor*.

The *Relay* role does not perform role assignment since it has no role management responsibility as shown in the mission class specification in Figure 6.8. The same applies to the *Hazard-detector*, *Medic* and *Transporter* roles. In addition, with respect to team-wide interactions these roles mainly receive orders to provide assistance, transport, etc. which require little or no coordination. Consequently, these roles have a smaller and less complex set of policies. Although there are no additional types of policies (compared to the policies considered in this chapter) introduced in these roles, for the sake of completeness, they are included in Appendix B.

The *Surveyor* role, on the other hand, searches for survivors and organises the rescue of survivors resulting in a complex set of policies which we will discuss in the following. The first set of policies of the *Surveyor* shown in Figure 6.19, however, are similar (with one significant difference) to those of the *Aggregator* role in that they configure the role, activate the role assignment (for the *Medic* and *Transporter* roles) policies, set optimisation parameters, start the discovery service and import references to assigned (managed) roles for future policy-based operation invocation. Unlike the *Aggregator* role, the *Surveyor* imports (lines 7 - 10) a reference to its manager (the *Aggregator* role), which is necessary if its policies need to invoke operations on the *Aggregator*'s interface as we will see later on. As mentioned before, both the *Commander* and *Aggregator* use policies to import references to their managed roles (Figure 6.15, lines 45 - 55 & Figure 6.17, lines 52 - 60 respectively) and set the state update type to *management*, which makes the managed roles (*Relay* and *Aggregator*, and *Surveyor* and *Hazard-detector*) include (in the state update messages) information, i.e., role identity objects pertaining to themselves and their managed roles, which enables the manager role to maintain the domain structure. Recall that the domain structure is used for maintaining collaboration organisation structures.

```
1  //Surveyor role startup policy
2  policy := root/factory/ecapolicy create.
3  policy event: /event/newSurveyor;
4  action: [ :name :role :instance
5  :parentrole :parentinstance :parentaddress :parentpath|
6  //Import parent role reference.
7  root/role at: parentrole ifAbsent:[root/role at:parentrole put:
8      (domainFactory create). ].
9  (root/role resolve: parentrole) at: parentinstance put:
10 (root import: parentpath from: parentaddress).
11 root at: "discovery" put: (root/factory/discovery create).
12 root at: "optimiser" put: (root/factory/optimiser create).
13 root/discovery setRole:role instance: instance.
14 //Create tasks.
15 ((root/task asHash) has: "explore") ifFalse: [
16 root/task at: "explore" put:((root load: "dmrc.task.Explore") create)].
17 ((root/task asHash) has: "identifysurvivor") ifFalse: [
18 root/task at: "identifysurvivor" put:
19     ((root load: "dmrc.task.IdentifySurvivor") create)].
20 ((root/task asHash) has: "report") ifFalse: [
21         root/task at: "report" put:
22     ((root load: "dmrc.task.Report") create)].
23 //Configure tasks.
24 root/task/identifysurvivor bindExploreTask:(/root/task/explore).
25 root/task/explore bindCameraTask:(/root/task/camera).
26 root/task/explore bindMotionTask:(/root/task/motion).
27 root/task/explore bindBuildMapTask: (/root/task/buildmap).
28 //Bind tasks.
29 (root/role resolve: (role+"/"+instance))
30 bindExploreTask: (root/task/explore).
31 (root/role resolve: (role+"/"+instance))
32 bindIdentifySurvivorTask: (root/task/identifysurvivor).
33 (root/role resolve: (role+"/"+instance))
34 bindReportTask: (root/task/report).
35 //Activate role assignment policies.
36 root/policy/assignTransporterOpt active: true.
37 root/policy/assignMedicOpt active:true.
38 root/discovery ready.].
39 policy active: true. root/policy at: "surveyorStartup" put: policy.
40 //Role assignment policies
41 policy := root/factory/ecapolicy create.
42 policy event: /event/discoveryReady;
43 action: [root/discovery has:#("lifting" "GPS" "motion")  assign:"transporter"].
44 root/policy at: "assignTransporterOpt" put: policy. policy active: false.
45 policy := root/factory/ecapolicy create.
46 policy event: /event/discoveryReady;
47 action: [ root/discovery has:#("medical" "GPS" "motion")  assign:"medic"].
48 root/policy at: "assignMedicOpt" put: policy. policy active: false.
49 //Discovery policy
50 policy := root/factory/ecapolicy create.
51 policy event: /event/newUAV;
52 condition:[root/discovery notAllAssigned.];
53 action: [ :lowLevelCap :uav |
54 root/discovery adduav:uav cap:lowLevelCap].
55 root/policy at: "addUav" put: policy. policy active: true.
56 //Optimisation policies
57 policy := root/factory/ecapolicy create.
58 policy event: /event/newSurveyor;
59 action: [root/utility/aggregate utility:"matching" setweight:1].
60 root/policy at: "matchingWeight" put: policy. policy active: true.
61 policy := root/factory/ecapolicy create.
62 policy event: /event/discoveryReady;
63 action: [root/optimiser setoptimrate: 20000].
64 root/policy at: "optimRate" put: policy. policy active: true.
65 policy := root/factory/ecapolicy create.
66 domainFactory := root load: "Domain".
67 policy event: /event/roleAssigned;
68 action: [ :name :role :instance :address :path|
69 root/role at: role ifAbsent:[root/role at:role put:(domainFactory create). ].
70 (root/role resolve: role) at: instance put:(root import: path from: address).].
71 root/policy at: "importRole" put: policy. policy active: true.
```

Figure 6.19: Search & Rescue Mission – Surveyor Role Policies (Part 1)

Figure 6.20 shows the collaboration organisation structure maintained by the search and rescue team in accordance with the aforementioned policies.

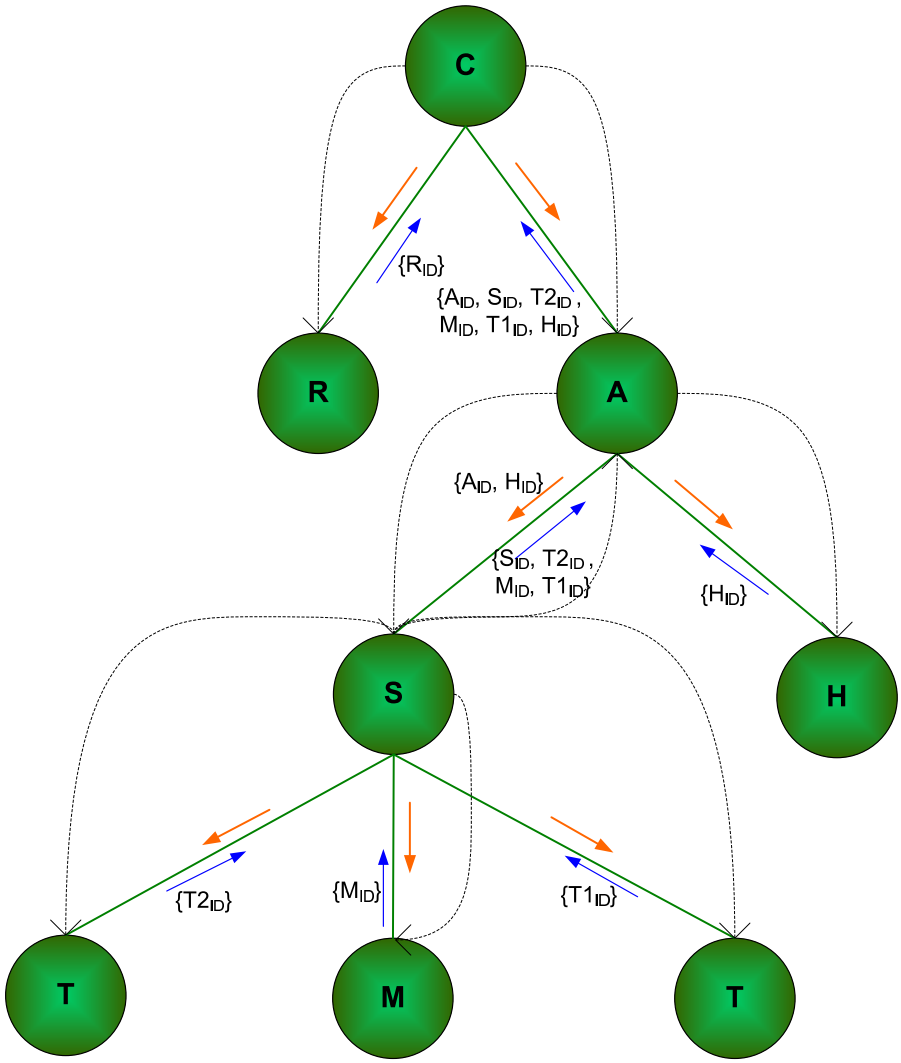


Figure 6.20: Search & Rescue Collaboration Organisation Structure

The broken lines show the collaboration with the line starting from the role that maintains the reference and ending with an arrow at the role to which the reference relates to. The solid lines show the management tree and the upward and downward state updates. The upward state updates contain role identity objects, which are used to maintain the collaboration organisation structure, as well as the velocity and location of the UAVs (not shown in the figure) while the downward messages contain only the velocity and location of UAVs (not shown in the figure). For example, the state update message sent by the *Hazard-detector* to its manager role (the *Aggregator*) contains H_{ID}, which is the *Hazard-detector*'s identity consisting of its role type, identity number, path and address. Note that the *Surveyor*, unlike the other manager

roles, has a reference to its manager role (the *Aggregator*) since it has a policy that imports a reference to its manager role. Also, none of the non-manager roles have a reference to any role since they do not have policies to do so. Should non-manager roles need to import and maintain references to their manager roles (e.g., the *Medic* role maintaining a reference to the *Surveyor* role), only adding a policy that imports the reference during the non-manager role's startup suffices since the role assignment message contains the necessary information for importing the reference and the *alive* messages are sufficient for maintenance. As mentioned before, the decision of whether to import remote role references is made by the administrator based on the roles' interactions and dictated by policies.

Using its discovery and assignment policies, the *Surveyor* role performs the *Medic* and *Transporter* role assignments, which completes the team formation resulting in the team shown previously in Figure 6.10. Since it may be the case that not enough UAVs are available during the mission startup, the team may take time to reach the complete configuration shown in the figure and hence the mission should adapt to availability of UAVs. In addition, on the way to or after achieving the complete configuration the team may keep changing due to (1) the mobile nature of UAVs resulting in some members departing and (2) failure of UAVs or communication links. The rescue strategy also may change through time depending on the result of the disaster and risk analysis such as whether the damage is uniform across the mission area and/or further collapse is prominent. In the next section, we will show how the mission adapts to changes in context such as availability of UAVs as well as the rescue strategy using the *Surveyor* role's policies, which dictate its interaction with the *Medic* and *Transporter* roles.

6.9 Search and Rescue Mission Adaptation

The *Surveyor* role, as it can be seen from the outline of role interactions (Figure 6.4), is responsible for coordinating the rescue of a survivor once the survivor is detected by the *Surveyor* role itself and the risk is assessed by the *Aggregator* role with the help of the report gathered from the *Hazard-detector* and *Surveyor* roles. In the previous section, we have seen a subset of the *Surveyor*'s policies that are related to discovery and assignment. In this section, we will consider more policies that dictate

the behaviour of the role depending on current context. In the following, we will present these policies part by part to make them manageable for presentation.

The first policy, $assessRisk$, shown in Figure 6.21 (lines 74 - 82) is triggered by the $survivorDetected$ event, which is generated by the *Surveyor* role itself as a result of its search for survivors. This event starts a chain of rescue-related operations, the first of which is risk assessment performed by the *Aggregator*. The *Aggregator*, upon performing the risk assessment, generates the $riskAssessed$ event, which triggers some out of a number of policies depending on the rescue strategy (considered later on). The next two policies, $reAssignTransporter$ and $reAssignMedic$ (lines 84 - 102) relate to managing the failure of the *Transporter* and *Medic* roles respectively. As discussed in Chapter 4, the management framework attempts to resolve UAV failures transparently by reassigning roles on failed UAVs to idle (previously discovered but not assigned) or newly discovered UAVs. However, when there are no idle or newly discovered UAVs, the response depends on the failure management policies. The *Transporter* failure policy (lines 84 - 92) withdraws any role (other than the *Transporter*) that is managed by this role (the *Surveyor*) and satisfies the capability requirements of the *Transporter* role, and reassigns it to a *Transporter* role. The *scheme* (line 90) indicates (1) which role to withdraw when there are multiple role types and/or instances that satisfy the requirement and/or (2) which failed *Transporter* to reassign when multiple *Transporter* roles have failed. Although we only have one scheme, namely *default* in the prototype implementation, we have factored out this behaviour to be set by a policy envisaging the addition of other schemes in future work. In the *default* scheme, the role with a lower role ID is withdrawn (assuming that it is new to the mission and removing it causes less disruption), and if more than one role has failed the one that failed first will be reassigned.

The *Medic* role failure management policy has the same behaviour as the *Transporter*'s policy. Since the *Surveyor* role manages only these two types of roles, when a *Transporter* role fails, if the *Transporter* failure management policy is active then a *Medic* role will be withdrawn and reassigned to a *Transporter* role and vice versa. Consequently, only one of the failure management policies should be active to avoid oscillation in the event of frequent simultaneous failures of the two roles. The decision as to which policy to activate depends on the strategy of the rescue, i.e., whether it focuses on providing medical assistance first or transporting

the survivors out of the rubble. We will later on come back to these policies when we discuss mission adaptation based on the rescue strategy. The remaining two policies, *surveyorBatteryLowSearchPattern* and *surveyorBatteryLowPicture* (lines 104 - 122) enable the *Surveyor* role to adapt its search pattern and visual services to the current context, i.e., the level of battery power and the strength of the wireless communication link radio signal.

```
72 //When a survivor is detected, perform risk assessment
73 //using the Aggregator's 'assessRisk' operation.
74 policy:=root/factory/ecapolicy create.
75 policy event: /event/survivorDetected;
76 condition:[(root/role/aggregator size) > 0];
77 action: [:survivorLocation :name|
78 aggregator := (root/role/aggregator listObjects) at: 0.
79 aggregator assessRisk: survivorLocation.
80 ].
81 root/policy at: "assessRisk" put: policy.
82 policy active: true.
83 //Transporter failure policy
84 policy := root/factory/ecapolicy create.
85 policy event: /event/UAVFailure;
86 condition:[:role| (role=="transporter") & (((root/role asHash) has: "medic")
87                ifFalse: [false] ifTrue: [((root/role/medic size) > 0)])];
88 action: [ :role :name :parentrole :parentinstance|
89 (root/role resolve: (parentrole+"/"+parentinstance)) reassign:
90 role scheme:"default"].
91 root/policy/common at: "reAssignTransporter" put: policy.
92 policy active: false.
93 //Medic failure policy
94 policy := root/factory/ecapolicy create.
95 policy event: /event/UAVFailure;
96 condition:[:role| (role=="medic") & (((root/role asHash) has: "transporter")
97        ifFalse: [false] ifTrue: [((root/role/transporter size) > 0)])];
98 action: [ :role :name :parentrole :parentinstance|
99 (root/role resolve: (parentrole+"/"+parentinstance)) reassign:
100 role scheme:"default"].
101 root/policy/common at: "reAssignMedic" put: policy.
102 policy active: false.
103 //When the battery power gets low, change the search pattern
104 policy := root/factory/ecapolicy create.
105 policy event: /event/batteryLevel;
106 condition:[:level :role :instance |
107        (level >= (root/role resolve: (role+"/"+instance)) missionParam:
108  "minBatteryLevel")) &
109        (level < 3*((root/role resolve: (role+"/"+instance)) missionParam:
110  "minBatteryLevel"))];
111 action: [ :name :role :instance|
112 (root/role resolve: (role+"/"+instance)) setSearchPattern: "random".].
113 root/policy at: "surveyorBatteryLowSearchPattern" put: policy.
114 policy active: true.
115 //Send a lower quality picture when the signal strength is low.
116 policy := root/factory/ecapolicy create.
117 policy event: /event/comStatus;
118 condition:[:level :role :instance :strength | (level < 4)];
119 action: [ :name :role :instance|
120 (root/role resolve: (role+"/"+instance)) setPictureQuality: "low".].
121 root/policy at: "surveyorBatteryLowPicture" put: policy.
122 policy active: true.
```

Figure 6.21: Search & Rescue Mission – Surveyor Role Policies (Part 2)

Recalling the chain of actions started by the *survivorDetected* event, we have previously seen that the *Surveyor* role's *assessRisk* policy invokes a risk assessment

operation on the *Aggregator* role. The *Aggregator* role, upon performing the risk assessment, generates a $riskAssessed$ notification, which triggers policies relating to enlisting (adding to a queue of survivors maintained by the *Surveyor* role) a survivor for future medical assistance and transport, or provision of immediate medical assistance and transport depending on the rescue strategy and other current context.

Consider a situation where the damage across the disaster area is nonuniform resulting in a rescue strategy where survivors (identified by their location) facing prominent danger (due to hazardous materials, likely collapse, etc.) are given higher priority for getting medical assistance and transport to the care centre. The policies shown in Figures 6.22 - 6.25 relate to the behaviour of the mission with respect to assistance and transport of survivors.

The first policy, $priorityAssistSingleMedic$, shown in Figure 6.22 (lines 124 - 139) responds to the $riskAssessed$ event and is relevant when there is only one *Medic* role and the risk assessment categorises the survivor as *injured* (lines 126 - 127). Provided that these conditions are satisfied, the priority of the survivor is computed (by the *Surveyor* role, line 132) and if the survivor has a *high* priority, the $idle$ operation of the *Medic* role is invoked to check whether the *Medic* role is idle or engaged. If the *Medic* role is idle its $assist$ operation, with the survivor location as a parameter, is invoked (line 137). If the *Medic* role is assisting another survivor or if the priority of the survivor is not *high*, the survivor is added to a waiting list for later assistance (line 134). When the number of *Medic* roles is more than one, the second policy, $priorityAssistMutipleMedic$, becomes relevant. In a similar manner to the single-medic priority-based policy, if the survivor does not have a *high* priority the corresponding action is adding the survivor to a waiting list for later assistance (line 150). However, if the survivor has a *high* priority the $measureMetric$ (lines 153 - 154) and $idle$ (lines 157 - 158) operations are invoked on all *Medic* roles. The engaged *Medic* roles are then filtered out (lines 163 - 166). Depending on the result of the filtering, if there is only one idle *Medic*, the $assist$ method is invoked on this *Medic*. If there is no idle *Medic* at all, the survivor is enlisted for later assistance (lines 169 - 172). If there are more than one idle *Medic*s, the nearest to the survivor is selected based on the measured metric, which is distance in this case (lines 180 - 186), and the $assist$ operation is invoked on the selected *Medic* (line 188). Both policies are presented with comments.

```
123 //Single Medic priority-based assistance
124 policy:=root/factory/ecapolicy create.
125 policy event: /event/riskAssessed;
126 condition: [:riskLevel| (((root/role asHash) has: "medic") ifFalse: [false]
127         ifTrue: [((root/role/medic size) == 1)]) & (riskLevel=="injured") ];
128 action: [:survivorLocation :name|
129 //Compute the priority of this survivor.
130 surveyor := (root/role/surveyor listObjects) at: 0.
131 medic := (root/role/medic listObjects) at: 0.
132 priority:= surveyor computePriority: survivorLocation risk: riskLevel.
133 ((priority == "high") & (medic idle.)) ifFalse: [
134 surveyor queue: "assist" add: survivorLocation  priority: priority] ifTrue: [
135 medic := (root/role/medic listObjects) at: 0.
136 //Invoke the 'assist' operation on the medic role.
137 medic assist: survivorLocation.].].
138 root/policy/nonuniform at: "priorityAssistSingleMedic" put: policy.
139 policy active: true.
140 //Multiple Medic priority-based assistance
141 policy:=root/factory/ecapolicy create.
142 policy event: /event/riskAssessed;
143  condition: [:riskLevel| (((root/role asHash) has: "medic") ifFalse: [false]
144 ifTrue: [((root/role/medic size) > 1)]) & (riskLevel=="injured") ];
145  action: [:survivorLocation :name|
146 //Compute the priority of this survivor.
147 surveyor := (root/role/surveyor listObjects) at: 0.
148 priority:= surveyor computePriority: survivorLocation risk: riskLevel.
149 (priority == "high") ifFalse: [
150 surveyor queue: "assist" add: survivorLocation priority: priority] ifTrue: [
151 //Invoke the 'measureMetric:metricType:' operation on all medic roles.
152 //The result is an array containing all the replies (results).
153 distance := (root/role/medic collect: [
154     :name :medic| medic measureMetric: "distance"]).
155 //Invoke the 'idle' operation, which has a boolean result depending on
156 //whether the medic role is assisting a survivor or not, on all medic roles.
157 status :=  (root/role/medic collect: [
158     :name :medic| medic idle.]).
159 //Get a copy of the reference (external interface) for each of the
160 //medic roles, in an array.
161 medics :=(root/role/medic collect: [:name :value| value]).
162 //Filter out the non-idle medic roles.
163 index := 0.
164 status do: [:value|(value) ifFalse: [
165     medics remove: index. distance remove: index]
166 ifTrue: [index.].  index := index+1. ].
167 //If all medics are engaged, enlist the survivor for later assistance
168 //if there is only a single idle medic role then use that.
169 ((medics size) == 0) ifTrue: [
170 surveyor queue: "assist" add: survivorLocation priority: priority ]
171 ifFalse: [ ((medics size) == 1) ifTrue: [(medics at: 0) assist: survivorLocation]
172 ifFalse: [
173 //otherwise select the medic with the smallest distance (nearest) to the
174 //survivor while(the size of the result array > 1){
175 //Compare the result at index 0 with the result at the last index
176 //remove the greater from the results list, remove the medic role
177 //reference at the same index as the greater result
178 //}
179 //the remaining medic is the one with the smallest distance (nearest)
180 [((distance size) >1)] whileTrue: [
181  ((distance at:0) < (distance at:((distance size)-1))) ifTrue: [
182     distance remove: ((distance size)-1).
183     medics remove: ((medics size)-1).]
184  ifFalse: [distance remove:0. medics remove:0.].].
185 //Get the role reference.
186 selectedMedic:=medics at:0.
187 //Invoke the 'assist' operation on the selected medic role.
188 selectedMedic assist: survivorLocation.
189 ].].].].
190 root/policy/nonuniform at: "priorityAssistMutipleMedic" put: policy.
191 policy active: true.
```

Figure 6.22: Search & Rescue Mission – Surveyor Role Policies (Part 3)

```
192 //Single Transporter priority-based policy
193 policy:=root/factory/ecapolicy create.
194 policy event: /event/survivorAssisted;
195 condition: [ ((root/role asHash) has: "transporter") ifFalse: [false]
196         ifTrue: [((root/role/transporter size) == 1)]];
197 action: [:survivorLocation :name|
198 transporter := (root/role/transporter listObjects) at: 0.
199 careCentre := ((root/role/surveyor listObjects)
200 at: 0)  missionParam: "careCentre".
201 surveyor := (root/role/surveyor listObjects) at: 0.
202 priority := surveyor getPriority: survivorLocation.
203 (priority=="high"& (transporter idle.)) ifFalse: [
204 surveyor queue: "transport" add: survivorLocation priority: priority] ifTrue: [
205 transporter transport: careCentre survivorLocation: survivorLocation.]].
206 root/policy/nonuniform at: "priorityTransportSingleTransporter" put: policy.
207 policy active: true.
```

Figure 6.23: Search & Rescue Mission – Surveyor Role Policies (Part 4)

Upon completion of providing the assistance, the *Medic* role generates the *survivor Assisted* notification, which triggers the single-transporter and multiple-transporter priority-based transport policies shown in Figures 6.23 & 6.24 respectively. These policies are similar in structure to the single-medic and multiple-medic priority-based assistance policies with one significant difference in the case of the multiple-transporter policy. In this policy, when there are more than one idle *Transporter*s to choose from, the selection is done by computing a weighted cost using distance (to survivor) and battery power, and then selecting the one with the minimum cost (lines 245- 272). Both policies are presented with comments. The priority-based single and multiple assistance and transport policies we have seen previously become relevant when there are at least one *Medic* and *Transporter* roles respectively. In the following, Figure 6.25, we consider policies that dictate the behaviour of the rescue mission when it is the case that no *Medic* and/or *Transporter* roles are available (e.g., roles not assigned yet, UAV departed or failed, etc.). In addition, policies that handle survivors that are enlisted for future assistance and transport are shown.

As shown in Figure 6.25, the *noMedicEnlistForAssistance* (lines 280-288) and *noTransporterEnlistForTransport* (lines 290-297), policies enlist the survivor for medical assistance and transport respectively.

The *directlyEnlistUninjuredForTransport* policy (lines 299 - 305), which is triggered by the *riskAssessed* event, unlike the other transport policies that are triggered by the *survivorAssisted* event, enlists survivors that do not need medical assistance for transport.

```
208 //Multiple Transporter priority-based policy
209 policy:=root/factory/ecapolicy create.
210 policy event: /event/survivorAssisted;
211 condition: [:riskLevel| (((root/role asHash) has: "transporter") ifFalse: [false]
212         ifTrue: [((root/role/transporter size) > 1)]) ];
213 action: [:survivorLocation :name|
214 //Compute the priority of this survivor.
215 surveyor := (root/role/surveyor listObjects) at: 0.
216 priority:= surveyor computePriority: survivorLocation risk: riskLevel.
217 (priority == "high") ifFalse: [
218         surveyor queue: "transport" add: survivorLocation priority: priority]
219 ifTrue: [
220 //Invoke the 'measureMetric:metricType:' operations on all transporter roles
221 //the result is an array containing all the replies (results).
222 distance := (root/role/transporter collect: [
223     :name :transporter| transporter measureMetric: "distance"]).
224 power :=  (root/role/transporter collect: [
225     :name :transporter| transporter measureMetric: "power"]).
226 //Invoke the 'idle' operation, which has a boolean result depending on
227 //whether the transporter role is transporting a survivor or not,
228 //on all transporter roles.
229 status :=  (root/role/transporter collect: [
230     :name :transporter| transporter idle.]).
231 //Get a copy of the reference (external interface) for each of the
232 //transporter roles, in an array.
233 transporters :=(root/role/transporter collect: [:name :value| value]).
234 //Filter out the non-idle transporter roles.
235 index := 0.
236 status do: [:value|(value) ifFalse: [
237     transporters remove: index. distance remove: index. power remove: index]
238         ifTrue: [index.].  index := index+1. ].
239 //If all transporters are engaged, enlist the survivor for later transportation
240 //if there is only a single idle transporter role then use that.
241 ((transporters size) == 0) ifTrue: [
242         surveyor queue: "transport" add: survivorLocation priority: priority ]
243 ifFalse: [ ((transporters size) == 1) ifTrue: [
244 (transporters at: 0) transport: careCentre survivorLocation: survivorLocation]
245 ifFalse: [
246 //Assign weights to distance and battery power depending
247 //on the distance from the survivor's location to the care centre.
248 ((surveyor distanceFrom: survivorLocation  To: careCenter) > 100 ) ifFalse: [
249 distanceWeight := 75. powerWeight := 25. ] ifTrue: [ distanceWeight := 25.
250 powerWeight :=75.]
251 //Find out the maximum battery power.
252 [((temp size) >1) ] whileTrue: [ (((temp at:0)> (temp at:((temp size)-1))))
253 ifTrue: [ temp remove: ((temp size)-1)] ifFalse: [temp remove: 0.] .].
254 maxPower := temp at: 0.
255 //while(the size of the result array > 1){
256 //compute the weighted cost of using a transporter role as
257 //distance_of_uav*distanceWeight + ((max - battery_power_of_uav)*powerWeight)
258 //where max = the maximum battery power among the received result from the
259 //measurement compare the result at index 0 with the result at the last index
260 //remove the greater from the results list, remove the transport role reference at
261 //the same index as the greater result
262 //}
263 //the remaining transporter is the one with the  smallest cost (highest utility)
264 [((distance size) >1) ] whileTrue: [
265  ((((distance at:0)*distanceWeight + (maxPower - (power at: 0))*powerWeight ))>
266         (((distance at: ((distance size) - 1) )*distanceWeight +
267                 (maxPower - (power at: ((power size) - 1) ))*powerWeight )))
268 ifTrue: [distance remove: ((distance size) - 1). power remove: ((power size) -1).
269           transporters remove: ((transporters size) -1).]
270 ifFalse: [ distance remove: 0. power remove: 0. transporters remove: 0.].].
271 //Get the role reference.
272 selectedTransporter:=transporters at:0.
273 //Invoke the 'transport' operation on the selected transporter role.
274 selectedTransporter transport: careCentre survivorLocation: survivorLocation.
275 ].].].].
276 root/policy/nonuniform at: "priorityTransportMutipleTransporter" put: policy.
277 policy active: true.
```

Figure 6.24: Search & Rescue Mission – Surveyor Role Policies (Part 5)

```
//No Medic, there is no domain entry for the Medic role (not assigned yet)  or
//the domain structure entry is there but the instance is not (due to failure).
policy := root/factory/ecapolicy create.
policy event: /event/riskAssessed;
 condition: [:riskLevel| (((root/role asHash) has: "medic")
        ifFalse: [(riskLevel=="injured")]
        ifTrue: [((root/role/medic size) == 0)]) & (riskLevel=="injured") ];
action: [ surveyor := (root/role/surveyor listObjects) at: 0.
surveyor queue: "assist" add: survivorLocation.].
root/policy/nonuniform at: "noMedicEnlistForAssistance" put: policy.
policy active: true.
//No Transporter (similar to the no medic policy)
policy := root/factory/ecapolicy create.
policy event: /event/survivorAssisted;
 condition: [:riskLevel| (((root/role asHash) has: "transporter")
        ifFalse: [true] ifTrue: [((root/role/transporter size) == 0)])];
action: [ surveyor := (root/role/surveyor listObjects) at: 0.
surveyor queue: "transport" add: survivorLocation.].
root/policy/nonuniform at: "noTransporterEnlistForTransport" put: policy.
policy active: true.
//Enlist survivors that are not injured for transport without medical assistance.
policy := root/factory/ecapolicy create.
policy event: /event/riskAssessed;
condition: [:riskLevel| (riskLevel != "injured") ];
action: [ surveyor := (root/role/surveyor listObjects) at: 0.
surveyor queue: "transport" add: survivorLocation.].
root/policy/common at: "directlyEnlistUninjuredForTransport" put: policy.
policy active: true.
//Assist survivors in the waiting list
//after completing assistance to another survivor.
policy:=root/factory/ecapolicy create.
policy event: /event/survivorAssisted;
condition: [ ((root/role/surveyor queueSize: "assist" ) > 0)];
action: [:role :instance|
medic := (root/role resolve: (role+"/"+instance) ).
surveyor := (root/role/surveyor listObjects) at: 0.
survivorLocation := surveyor queue: "assist"  remove.
medic assist: survivorLocation.]].
root/policy/common at: "assistNextAfterCompletion" put: policy.
policy active: true.
//Assist survivor in the waiting list if idle.
policy:=root/factory/ecapolicy create.
policy event: /event/stateUpdate;
condition: [ :role :status |(role=="medic")&(status=="idle")
       & ((root/role/surveyor queueSize: "assist") > 0)];
action: [:role :instance :status|
medic := (root/role resolve: (role+"/"+instance) ).
surveyor := (root/role/surveyor listObjects) at: 0.
survivorLocation := surveyor queue: "assist"  remove.
medic assist: survivorLocation.]].
root/policy/common at: "assistNext" put: policy. policy active: true.
//Transport survivor in the waiting list
//after completing transportation of another survivor.
policy:=root/factory/ecapolicy create.
policy event: /event/survivorTransported;
condition: [ ((root/role/surveyor queueSize: "transport" ) > 0)];
action: [:role :instance|
transporter := (root/role resolve: (role+"/"+instance) ).
surveyor := (root/role/surveyor listObjects) at: 0.
survivorLocation := surveyor queue: "transport"  remove.
transporter transport: careCentre survivorLocation: survivorLocation.]].
root/policy/common at: "transportNextAfterCompletion" put: policy.
policy active: true.
//Transport survivor in the waiting list if idle.
policy:=root/factory/ecapolicy create.
policy event: /event/stateUpdate;
condition: [ :role :status |((role=="transporter")&(status=="idle")
       & (root/role/surveyor queueSize: "transport" ) > 0)];
action: [:role :instance :status|
transporter := (root/role resolve: (role+"/"+instance) ).
surveyor := (root/role/surveyor listObjects) at: 0.
survivorLocation := surveyor queue: "transport"  remove.
transporter transport: careCentre survivorLocation: survivorLocation.]].
root/policy/common at: "transportNext" put: policy. policy active: true.
```

Figure 6.25: Search & Rescue Mission – Surveyor Role Policies (Part 6)

Since the *Surveyor* receives periodic state updates from its managed roles (i.e., *Medic* and *Transporter*), it generates the $stateUpdate$ notification when it detects change in the status (whether it is idle or not) of the *Medic* and/or *Transporter* role. This event triggers policies that invoke either the $assist$ or $transport$ operation, if there is a survivor waiting for these services, depending on whether the event relates to a *Medic* ($assistNext$ policy, lines 319 - 328) or *Transporter* ($transportNext$ policy, lines 342 - 351) role respectively.

Upon assisting or transporting a survivor, the *Medic* and *Transporter* roles generate $survivorAssisted$ and $survivorTransported$ notifications respectively which in turn may trigger policies relating to medical assistance ($assistNextAfterCompletion$ policy, lines 308 - 317) or transport ($transportNextAfterCompletion$ policy, lines 331 - 340) depending on the existence of more survivors in the waiting list.

Figure 6.26 illustrates the interaction between the *Surveyor*, and *Medic* and *Transporter* roles and the evolution and/or devolution of the sub-team as UAVs become available and/or depart/fail as specified by the policies shown in Figures 6.21 - 6.25. The labeled solid lines with arrows represent policies. The lines originate from the role that performs the invocation (the subject of the policy) and terminate with arrows at the role on which the invocation is made (the target). The labels, P_i, refer to the policies considered in the scenario (these subscripted aliases are used only for convenience in the diagrams). Table 6.2 shows the mapping of the aliases, used in the current (Figure 6.26) and later diagrams, to the path and name of the actual policies.

We will now add two more assistance and transport policies that will later on be used by adaptive policies that perform policy substitution. These policies, shown in Figure 6.27, enlist survivors for future transport ($enlistForAssistance$ policy, lines 353 - 358) and medical assistance ($enlistForTransport$ policy, lines 360 - 365) irrespective of the priority of the survivor and the number of available *Medic* and *Transporter* roles.

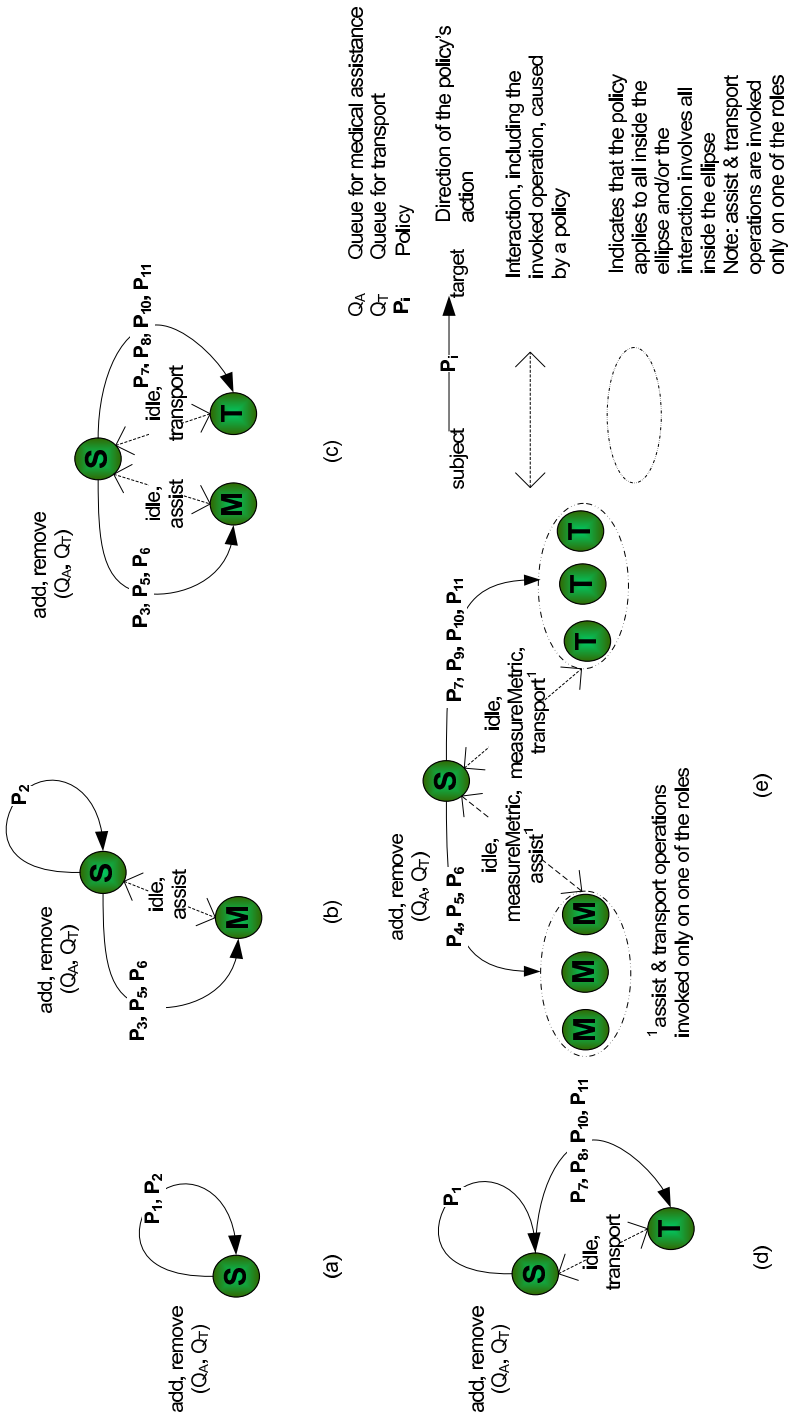


Figure 6.26: Priority-based Search & Rescue Mission

Alias	Policy
P_1	root/policy/nonuniform/noMedicEnlistForAssistance
P_2	root/policy/nonuniform/noTransporterEnlistForTransport
P_3	root/policy/nonuniform/priorityAssistSingleMedic
P_4	root/policy/nonuniform/priorityAssistMutipleMedic
P_5	root/policy/common/assistNextAfterCompletion
P_6	root/policy/common/assistNext
P_7	root/policy/common/directlyEnlistUninjuredForTransport
P_8	root/policy/nonuniform/priorityTransportSingleTransporter
P_9	root/policy/nonuniform/priorityTransportMutipleTransporter
P_{10}	root/policy/common/transportNextAfterCompletion
P_{11}	root/policy/common/transportNext
P_{12}	root/policy/uniform/enlistForAssistance
P_{13}	root/policy/uniform/enlistForTransport
P_{14}	root/policy/common/reAssignTransporter
P_{15}	root/policy/common/reAssignMedic
P_{16}	root/policy/uniformRiskHighCollapse
P_{17}	root/policy/uniformRiskLowCollapse
P_{18}	root/policy/nonUniformRisk

Table 6.2: List of Surveyor Role Policies

```
352 //Enlist survivor for medical assistance.
353 policy := root/factory/ecapolicy create.
354 policy event: /event/riskAssessed;
355 action: [
356 surveyor := (root/role/surveyor listObjects) at: 0.
357 surveyor queue: "assist" add: survivorLocation.].
358 root/policy/uniform at: "enlistForAssistance" put: policy. policy active: false.
359 //Enlist survivors who have received medical assistance for transport.
360 policy := root/factory/ecapolicy create.
361 policy event: /event/survivorAssisted;
362 action: [
363 surveyor := (root/role/surveyor listObjects) at: 0.
364 surveyor queue: "transport" add: survivorLocation.].
365 root/policy/uniform at: "enlistForTransport" put: policy. policy active: false.
```

Figure 6.27: Search & Rescue Mission – Surveyor Role Policies (Part 7)

The policies we have seen so far enable adaptation of the mission to changes in context. It is also possible to achieve multiple levels of adaptation by employing policy substitution that define ranges of adaptive behaviours for different strategies (which themselves are context dependent). The rescue strategy we have considered so far is a priority-based one, which is suitable when the damage across the disaster

area is nonuniform and consequently some survivors need immediate rescue than the others. On the other hand, if the damage across the disaster area is uniform, a non-priority based rescue with more focus either on transport or medical assistance depending on whether further collapse is highly or less likely may be suitable. The damage pattern itself may not be known a priori and hence the rescue mission needs to self-adapt to different rescue strategies as it determines the pattern of the damage.

The *Commander* role, using information gathered from all the other roles determines the damage pattern and generates the $riskPattern$ notification when there is a change in pattern. This event triggers one of the adaptation policies shown in Figure 6.28 which in turn activates/deactivates groups of policies depending on the damage pattern. All the policies that are activated/deactivated are discussed previously.

The first policy, $nonUniformRisk$ (lines 367 - 390) becomes relevant when the damage pattern is nonuniform (line 369) and employs a priority-based rescue strategy by deactivating all policies that do not apply to this strategy (lines 372 & 373) and activating all policies pertaining to this strategy (lines 375 - 378). It also selects a failure management policy depending on whether there are more survivors waiting for medical assistance or transport (lines 381 - 388). It is worth noting that although we show the activation/deactivation of policies one by one for the sake of clarity, the policies that apply to different strategies are grouped in a domain structure and hence can be collectively activated or deactivated succinctly. For example, all the priority-based (nonuniform pattern) or the non-priority-based (uniform pattern) policies can be activated as follows:

```
(root/policy resolve: pattern) do: [:value| value active: true].
```

The second policy, $uniformRiskHighCollapse$, (lines 392 - 410) becomes relevant when the damage pattern is uniform and further collapse is highly likely (line 394). It deactivates all policies that do not apply to this context (lines 397 - 400) and activates the non-priority based assistance and transport policies (lines 402 & 403). Since collapse is highly likely in this context, more weight is given to transporting survivors out of the disaster area than assisting them on the spot and hence the failure management strategy is an immediate reassignment of a *Transporter* (resulting in withdrawal of a *Medic* role) when it fails. Consequently, this policy deactivates the *Medic* failure policy and activates the *Transporter* failure policy (lines 406 & 407).

```
366 //Priority-based rescue strategy
367 policy := root/factory/ecapolicy create.
368 policy event: /event/riskPattern;
369 condition: [:pattern :name| pattern == "nonuniform"];
370 action: [
371 //Deactivate policies that do not apply to this risk pattern.
372 root/policy/uniform/enlistForAssistance active: false.
373 root/policy/uniform/enlistForTransport active: false.
374 //Activate policies that apply to this risk pattern.
375 root/policy/nonuniform/priorityAssistSingleMedic active: true.
376 root/policy/nonuniform/priorityAssistMutipleMedic active: true.
377 root/policy/nonuniform/priorityTransportSingleTransporter active: true.
378 root/policy/nonuniform/priorityTransportMutipleTransporter active: true.
379 //Activate/deactivate failure management policies depending on whether
380 //there are more survivors waiting for medical assistance or transport.
381 ((root/role/surveyor queueSize: "assist") >
382         (root/role/surveyor queueSize: "transport")) ifTrue: [
383 root/policy/common/reAssignMedic active: true.
384 root/policy/common/reAssignTransporter active: false.
385 ] ifFalse:[
386 root/policy/common/reAssignMedic active: false.
387 root/policy/common/reAssignTransporter active: true.]
388 ].
389 root/policy at: "nonUniformRisk" put: policy.
390 policy active: true.
391 //Non-priority-based rescue strategy, with likely collapse.
392 policy := root/factory/ecapolicy create.
393 policy event: /event/riskPattern;
394 condition: [:pattern :name| pattern == "uniform"  & collapse == "high"];
395 action: [
396 //Deactivate policies that do not apply to this risk pattern.
397 root/policy/nonuniform/priorityAssistSingleMedic active: false.
398 root/policy/nonuniform/priorityAssistMutipleMedic active: false.
399 root/policy/nonuniform/priorityTransportSingleTransporter active: false.
400 root/policy/nonuniform/priorityTransportMutipleTransporter active: false.
401 //Activate policies that apply to this risk pattern.
402 root/policy/uniform/enlistForAssistance active: true.
403 root/policy/uniform/enlistForTransport active: true.
404 //Activate Transporter failure management
405 //deactivate Medic failure management
406 root/policy/common/reAssignMedic active: false.
407 root/policy/common/reAssignTransporter active: true.
408 ].
409 root/policy at: "uniformRiskHighCollapse" put: policy.
410 policy active: true.
411 //Non-priority-based rescue strategy with less likely collapse.
412 policy := root/factory/ecapolicy create.
413 policy event: /event/riskPattern;
414 condition: [:pattern :name| pattern == "uniform"  & collapse == "low"];
415 action: [
416 //Deactivate policies that do not apply to this risk pattern.
417 root/policy/nonuniform/priorityAssistSingleMedic active: false.
418 root/policy/nonuniform/priorityAssistMutipleMedic active: false.
419 root/policy/nonuniform/priorityTransportSingleTransporter active: false.
420 root/policy/nonuniform/priorityTransportMutipleTransporter active: false.
421 //Activate policies that apply to this risk pattern.
422 root/policy/uniform/enlistForAssistance active: true.
423 root/policy/uniform/enlistForTransport active: true.
424 //Activate/deactivate failure management policies.
425 root/policy/common/reAssignMedic active: true.
426 root/policy/common/reAssignTransporter active: false.
427 ].
428 root/policy at: "uniformRiskLowCollapse" put: policy.
429 policy active: true.
```

Figure 6.28: Search & Rescue Mission – Surveyor Role Policies (Part 8)

The third policy, $uniformRiskLowCollapse$, (lines 412 - 429) becomes relevant when the damage pattern is uniform but collapse is less likely. In a similar manner to

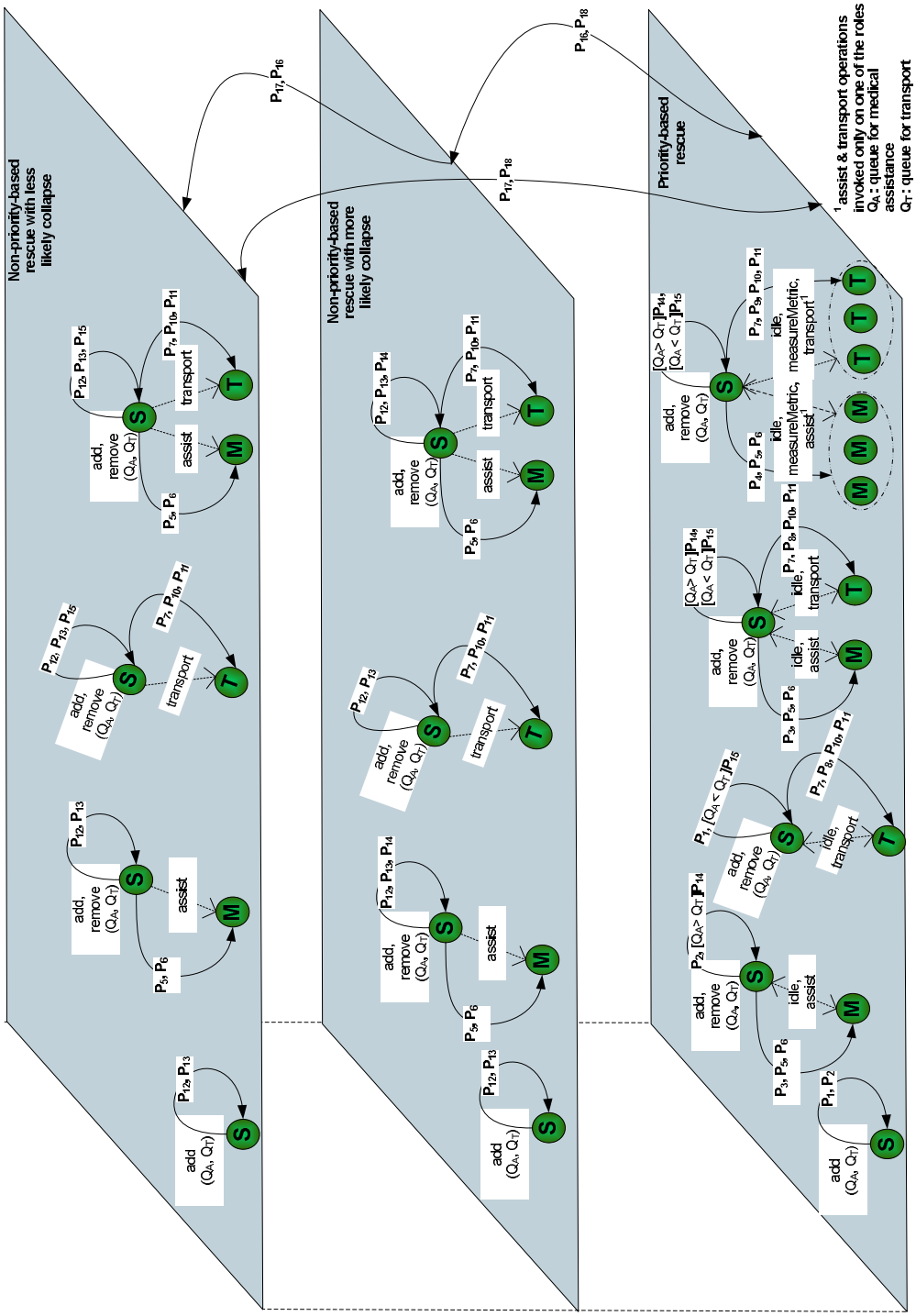


Figure 6.29: Search & Rescue Mission – Multiple Levels of Adaptation

the $uniformRiskHighCollapse$ policy, it deactivates all policies that do not apply to this context and activates the non-priority-based assistance and transport policies. It, however, gives more weight to medical assistance on the spot as collapse is less likely. Consequently, it deactivates the *Transporter* failure policy and activates the *Medic* failure policy (lines 425 & 426).

The three policies $nonUniformRisk$, $uniformRiskLowCollapse$ and $uniformRiskHigh Collapse$ select a mission behaviour that is suitable to the current context with respect to the damage pattern. The selected mission behaviour itself adapts to current context as we have seen previously. Consequently, the search and rescue mission is capable of achieving multiple levels of adaptation as shown in Figure 6.29.

The semantics of the symbols used in this diagram (Figure 6.29) is as described before in Figure 6.26 with one additional expression $[condition]P_i$ which means that policy P_i is activated when $condition$ is true. The actual policies represented by the subscripted aliases are also shown before in Table 6.2.

6.10 Conclusion

In this chapter, we have used a search and rescue scenario to show how a mission for a group of UAVs can be specified in terms of roles and how the role behaviours can be specified using policies giving rise to a reusable mission specification and an adaptable mission.

By using a set of policies to achieve a first level of adaptation to current context including the availability of UAVs, battery power, communication link, departure or failure of UAVs, and another set of policies to substitute these adaptation policies depending on another context, multiple levels of adaptation were illustrated.

Although we have illustrated the mission design process starting from scratch, we envision a trend where roles, tasks, policies and mission class specifications are stored in a repository and the mission administrator fetches suitable specifications for the mission statements at hand and use them with little or no adaptation.

The mission refinement in this chapter was done manually; tool support for this task would enhance the framework. Work in this direction has been done in [BLMR04,

BLR+06]. The work in refinement is continuing in the UK Ministry of Defence (MoD) and US Army Research Lab (ARL) funded International Technology Alliance (ITA) consortium.

Chapter 7

Implementation

In this chapter, we will present details of our prototype implementation of the self-management framework. The framework is built by extending the implementation of the self-managed cell [LDS+08] system, which has an asynchronous event bus and a policy service.

7.1 Overview of the Implementation

The management framework is designed and implemented as a composition of interacting entities organised in three layers, i.e., mission, team and communication. It is implemented using the Java-based Ponder2 [Pon] policy toolkit with most of the entities implemented as Ponder2 managed objects and stored in a domain[1] structure. An outline of the framework is shown in Chapter 3 (Figure 3.2). Although the majority of interactions take place between the entities inside the same or adjacent layer, the layering is not strict in that bypassing a layer to interact with entities in a non-adjacent layer is allowed.

In the following sections, we present a brief overview of the domain structure that is used to store entities of the framework followed by implementation details of the different framework elements. A note on managed objects is presented in Appendix B (Section B.4). Class diagrams are shown in Appendix C.

[1] Domains are similar to directories.

7.2 Domain structure

The management framework stores controllers, tasks, policies, roles and other entities in a domain structure as shown in Figure 7.1(a). Any entity in the domain structure can be loaded, enabled, disabled, or removed at run time using policies. A UAV stores two types of role objects or role-object references, namely local and remote where the role object representing the role to which the UAV has been assigned is local and references to the roles (role objects) enacted by collaborating UAVs are remote. The domain structure shown in Figure 7.1(b) is the *role* sub-domain of a UAV enacting the *Surveyor* role in the search and rescue mission (Chapter 6). Hence, it contains the *Surveyor* role object (local interface for local policies), the external interface of the *Surveyor* role (for remote roles' policies) and references (external interfaces) to the *Aggregator* (its manager role) as well as *Transporter* and *Medic* roles (its managed roles), which are enacted by other UAVs. In the example domain structure, note the absence of domain entries for other roles of the search and rescue mission such as the *Commander*. This is because the *Surveyor* policies used to import remote roles (Figure 6.19, lines 7 - 10 and lines 67 - 71) do so only for the manager (*Aggregator*), and managed (*Medic* & *Transporter*) roles since the *Surveyor* interacts only with these roles. This selective and policy-based approach to importing role references enables the mission administrator to control and minimise the domain-structure maintenance overhead. Figure 7.1(c) shows the *role* sub-domain structure for the *Aggregator* role in the same search and rescue mission.

A UAV bootstraps the framework by creating the domain structure used to store framework and mission-dependent elements. It then loads framework elements. Mission-dependent elements are generally loaded after a UAV is assigned to a role. The bootstrapping PonderTalk commands are shown in Appendix B (Figure B.8). The underlying mechanism for accessing a remote role's operations is Java's RMI (Remote Method Invocation). Hence, all UAVs maintain an RMI registry.

7.3 Mission Layer

The mission layer consists of three main entities, namely *Mission*, *Role Manager* and *Role*. The *Mission* entity deals with parsing the XML mission class specification and

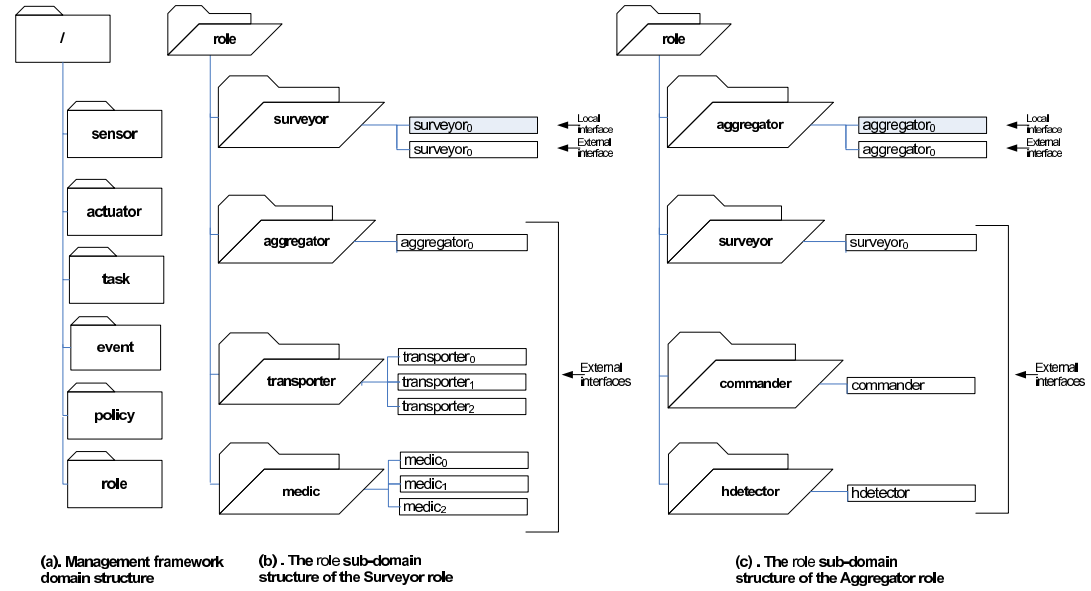


Figure 7.1: Domain Structure

instantiating the mission using the mission-class instance specification. The mission class and instance specifications may be provided directly to the *Mission* entity or they can be fetched from a repository if the mission administrator provides a URI.

Loading a Mission

The *Mission* entity is implemented by the $MissionClass$ class. This class provides methods to load, instantiate and get a sub-mission instance. In the remaining part of this section, we will refer to the $MissionClass$ class as *Mission entity*, since the *Mission* entity is implemented by this single class there will be no loss of semantics. A mission administrator, from a management console, or the commander role through its mission-loading policy invokes the **loadMissionClassSpec** method to load the mission class specification.

loadMissionClassSpec(String mcSpec, boolean isURI)

The $mcSpec$ parameter can contain either the mission class specification itself in XML format or a URI to the mission class specification. The $isURI$ flag selects the input type.

The *Mission* entity recursively builds a role tree in accordance with the mission class specification using the *Commander* role as the root node of the tree. The role tree is

stored using a nested instance of Java's *Hashtable* data structure. It contains mission parameters, roles and place holders for the associated role missions, structured hierarchically based on the management relation specified in the mission class specification. The tree does not contain the actual role-missions as these are specified in the mission class instantiation specification. The mission class is instantiated using the **instantiateMissionClas** method.

instantiateMissionClass(String mciSpec, boolean isURI, boolean fetchAll)

This method populates the role tree formed during the mission class loading with values from the mission-class instance specification. The $fetchAll$ flag instructs the method either to fetch and put the actual role-missions (policies) in the role-mission place holder or put URI's (values from the mission instance specification) and let the roles fetch their own role-mission later on. The *Commander* role implements the $mcspec : instspec$: PonderTalk message that invokes these two methods, with the input selection flag set to $true$ for both methods[2].

@Ponder2op("mcspec:instspec:")
loadMission(String mcSpec, String mciSpec)

This operation can be used to load the mission (both class and instance specifications) using a policy and a URI as follows (only the action part of the policy is shown):

```
root/role/commander/commander
mcspec:"http://192.168.0.1/mission_class_spec_case_study"
instspec:"http://192.168.0.1/mission_class_instance_spec_case_study"
fetchAll: false.
```

During role assignment the manager role, i.e., the role performing the assignment, requests the *Mission* entity for a mission instance using the **getSubMcInstance** method.

getSubMcInstance(String role)

Upon receiving the request, the *Mission* entity forms a subtree of the role tree where the role to be assigned is the root of the subtree and replies to the manager role with the subtree and mission parameters.

[2]For the relation between PonderTalk operations, the $@Ponder2op$ annotation and Java methods in managed objects such as roles, see Appendix B, Section B.4.

When a UAV is assigned to a role, if the role is a *Commander* role, the *Role Manager* requests the *Mission* entity for a mission instance. For all other roles, the mission instance is received with the role assignment message. The *Role Manager* entity then retrieves the mission instance from the role assignment message and updates the *Mission* entity using the $setMissionClassInstance$ method.

setMissionClassInstance (Hashtable<String , Object> mcInstance)

The *Mission* entity can later be queried, during role assignment, for sub-mission instances if the role manages others roles through its **getSubMcInstance** method.

The Role Manager

The *Role Manager* entity, implemented by the $RoleManager$ class, is responsible for loading and withdrawing a role during mission startup and reconfiguration respectively. It receives messages from the UAV enacting the manager role and instantiates the role the local UAV is assigned to. The local UAV may already have the role code loaded on to the domain structure but not instantiated (see the management framework bootstrapping statements in Appendix B, Figure B.8), or the code may not be loaded but available locally. The $RoleManager$ can fetch the role code from the repository if the code for the assigned role type is unavailable in the local UAV.

The $RoleManager$ class provides the **loadrole** method, which is used to load a role.

loadRole (RoleAssignmentMessage roleAssignmentMessage)

The **RoleAssignmentMessage** class is a data structure that contains the identity (discussed later on) of the assignee role, the identity of the assigner role, a subset of the mission class instance (sub-mission instance) that is going to be managed by the role (note that this also may include role-mission policies). This data structure also has a place holder for another data structure, namely $State$, which is set to the failed role's state when the message is a role reassignment one.

Using the information from the role identity, the *Role Manager* creates two elements corresponding to the role – (1) the role object and (2) the associated adaptor object representing the role's external interface. It, then, creates a domain entry for the role type and puts these elements in the corresponding domain entry. Finally, it starts

the role by invoking the **start** method of the role, and sends a confirmation message to the manager role that sent the role assignment message. In the confirmation message, the path to the adaptor object representing the external interface is included as opposed to that of the role object. Should the manager role rely on services provided by this role, it imports the adaptor object to its domain structure, for use by its policies thereby creating a collaboration link (part of the collaboration organisation structure) between itself and the role.

Role

Roles are mission dependent and hence every mission has its own types of roles. However, all roles are subclasses of the abstract *Role* class, which implements the mission-independent management functions of a role such as role assignment. The mission-specific role classes implement PonderTalk messages that make some of these functionalities available to policy actions as well as other elements of the framework.

The **start** method of the *Role* class is used by the *Role Manager* to start the role, with the role assignment message received from the manager role (parent role), after completing the domain entry and role-object creation.

```
start(RoleAssignmentMessage roleAssignmentMessage)
```

When a role is instantiated, if it is a manager role, i.e., responsible for assigning other roles, it creates the necessary entities for team formation. It will then generate a waiting list for the roles it has to assign and provides this to the team layer, which uses it to optimise role assignments. A role can determine whether it is a manager role or not by checking the mission instance from the assignment message it receives through its **start** method from the *Role Manager*.

The **start** method creates an instance of the *ManagementTreeNode* and sets up the management relationship using the role identities retrieved from the role assignment message. It then retrieves the mission class instance from the role assignment message, and if this role is a manager role it creates a waiting list for assignment that can be retrieved by other framework elements through the **getWaitingRolesList** method.

```
@Ponder2op("getwaitinglist")
```

getWaitingRolesList()

The **start** method also creates an instance of the $StateAggregator$ using the failure timeouts from the mission class instance, loads its policies using the mission class instance and generates a $new[R]$ event, where R is the role type. This event triggers policies that load role-responsibility dependent (whether the role is a manager role or not) framework elements such as the discovery service and mission-specific elements such as tasks (e.g., the newAggregator event that triggers the policy shown in lines 3 - 21, Figure 6.17, which creates the discovery service). After this point, the role is fully functional. For example, it is able to discover UAVs and assign the roles it has put in the waiting list. The *State Aggregator* updates the waiting list during failure using the **addToWaitingRolesList** method.

addToWaitingRolesList(RoleIdentity roleId, State state)

$RoleIdentity$ and $State$ are data structures used to uniquely identify roles in the mission, and store and/or communicate the state of a role respectively.

The identity I of a role is defined as: $I = M : H : S$ where M = mission ID, H = hierarchy level and S = sequence number. The mission ID and hierarchy level are each represented using one byte. The sequence number, which uniquely identifies roles that are managed by the same manager role, is represented as an array of bytes with a variable length (depends on depth of the management hierarchy as specified by the mission class). Each role gets its identity from its manager role during role assignment. The mission ID is set through the mission specification (parameters or policy) with a default ID of 0. Both the hierarchy level and sequence number start from 0 and progress during assignment. Consequently, by default, the *Commander* role has an identity of $0 : 0 : 0$. Each manager role (including the *Commander* role) generates an identity for its managed roles as $M : H + 1 : IncrementAfter(S[H + 1])$ where $S[H + 1]$ is the $(H + 1)$'th byte of the sequence byte array. For example, for the search and rescue team of the case study (Figure 6.10), if the *Aggregator* role is assigned before the *Relay* and the *Surveyor* is assigned after the *Hazard-detector*, and one of the *Transporter*s is the last role to be assigned among those managed by the *Surveyor*, the role identities (numbers shown in hexadecimal) for the *Commander*, *Aggregator*, *Surveyor* and the last *Transporter* would be $00 : 00 : 000000$, $00 : 01 :$

010000, 00 : 02 : 01**01**00, 00 : 03 : 0101**05** respectively.

The *Discovery Service* requests the role for a role assignment message using the **getRoleAssignmentMessage** method.

```
@Ponder2op("role:uav:lowlevelcap:requirement")
getRoleAssignmentMessage(String roleType, String uav, String lowLevelCap,
    P2Array requirement)
```

The $roleType$ is the type of the role to be assigned, uav is the ID of the UAV that is going to enact this role, $lowLevelCap$ is the low-level capability description of the UAV encoded in XML and $requirement$ is the capabilities requirement of the role as set by the role assignment policy. This method prepares the role assignment by populating the role assignment message data structure with this role's identity (parent identity) the managed role's identity (child identity) and the sub-mission class instance corresponding to the managed role. The discovery service must already have used the uav, $lowLevelCap$ and $requirement$ to vet the UAV. These arguments are passed to this method so that the role stores them and uses them later in order to make immediate reassignment (by withdrawing another role) during failure (if there is a policy dictating to do so).

The **roleReassign** method is used by policies to enforce reassignment of a failed role as opposed to waiting for new UAVs to be discovered.

```
@Ponder2op("reassign:scheme:")
roleReassign(String failedRole, String scheme)
```

The method checks if one of the role-enacting UAVs satisfies the capabilities requirement of the $failedRole$. If so, it removes the role instance (external interface) enacted by this UAV from the domain structure (if the role's reference had been imported), removes the failed role with its state from the waiting list, prepares and sends a role assignment message to the UAV forcing it to withdraw the current role and enact the new role. It, then, adds the withdrawn role to the waiting list with its current state. Since the discovery service (unless it is turned off by a policy) checks the waiting list whenever a new UAV is discovered, the withdrawn role may eventually be assigned when (if) a new UAV is discovered. The discovery service, as well as other framework elements, can check if a role needs to perform an assignment of a given role type by

checking the waiting list using the **checkWaitingList** method.

```
@Ponder2op("checkWaitingList:")
checkWaitingRolesList(String roleType)
```

A role registers its interest in remote notifications by accessing the $Role$ class' $required Notifications$ list. Other roles check this list through the **isRequired (String eventName)** method provided by the role. The role also declares its exposed notifications by registering the exposed notification types in $exposedNotification$ list. These registrations are made for notifications defined as required and exposed in the role specification respectively. When events occur inside a role (in its tasks or itself), if the event is in the $exposedNotification$ list, the role checks whether other roles in the domain structure require this notification through the **isRequired** method of the external interfaces of the roles and invokes the **notify (P2Object event)** method on the external interface for all that require the event.

Tool Support

A tool that can be used to generate Java source code for the role and its corresponding external interface classes from the XML role specification is implemented. The main console of this tool is shown in Figure 7.2. The left-side panel shows the XML specification; the middle and right-side panels show the generated Java code for the role and its external interface respectively.

The role class generated by this tool contains all the common management operations and role-specific operations that are forwarded to the tasks of the role. The administrator can then add additional code depending on the role's mission-specific program logic. The external interface class, on the other hand is an adaptor class that can be used as it is, since the sole purpose of this class is forwarding calls from remote roles to the local instance of the role, and all the code relating to this functionality is generated from the role specification.

The main elements in the code generation of the role and its external interface are summarised in Table 7.1.

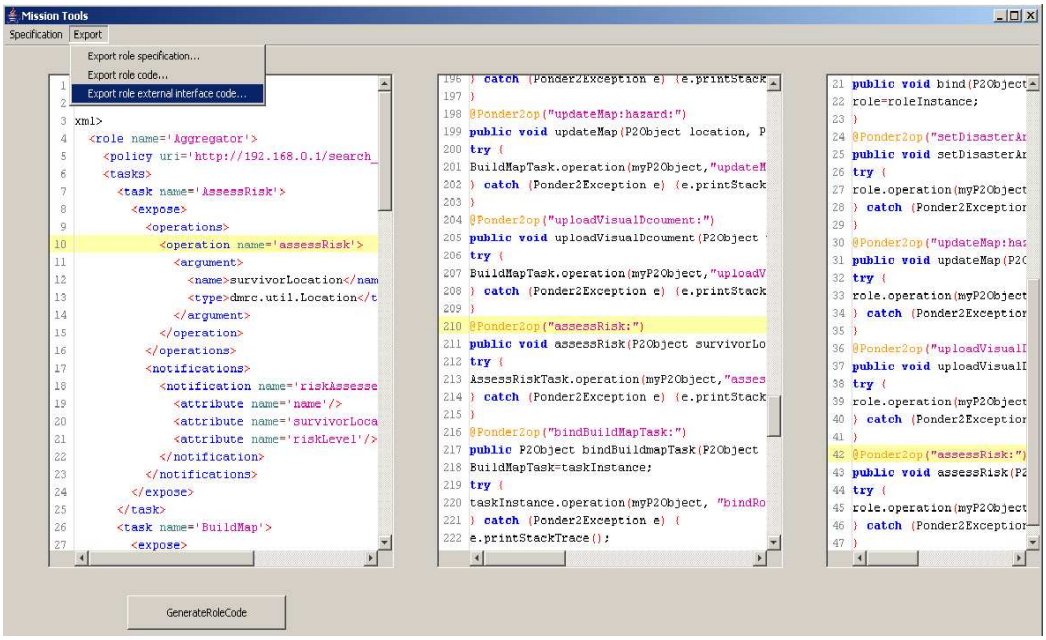


Figure 7.2: Role Specification and Code Generation Tool

7.4 Team Layer

The team layer has entities that are available in all UAVs irrespective of the role they are enacting: *Capability Manager*, *State Aggregator* and *Management Tree Node* as well as other entities whose existence depend on whether the UAV is a manager role or not: *Discovery* and *Optimiser*.

Management Tree and State Aggregation

The management tree is a distributed data structure used to maintain a UAV team. It is by means of this data structure that the framework represents and maintains the hierarchical management structure. The local component of this distributed structure, the *Management Tree Node* entity, which is responsible for keeping track of the state of the local role as well as roles managed by the local role is implemented by the $ManagementTreeNode$ data structure. The state information maintained by this entity is used for management purposes such as failure detection and role reassignments. The data structure mainly consists of the identity and state of the manager role (parent) of the node, the identities and states of the managed roles (children nodes) as well as the role's own state. The $ManagementTreeNode$ class provides a number of methods to add, retrieve and update children nodes (i.e., their identities)

Role Specification Element	**Class**	**Code**
All tasks	Mission-specific role	Add **Ponder2op** annotated task binding methods corresponding to each task in order to support PonderTalk operations that bind the task place holders in the **role** object to the **task** object in the domain structure.
	External interface of the role	None.
Exposed operations from tasks	Mission-specific role	Add a **Ponder2op** annotated method, corresponding to each exposed operation, which would accept a PonderTalk operation and forward the invocation to the **task** object.
	External interface of the role	Add a **Ponder2op** annotated method, corresponding to each exposed operation, which would accept a PonderTalk operation and forward the invocation to the **role** object.
Local operations from tasks	Mission-specific role	Add a **Ponder2op** annotated method, corresponding to each operation, which would accept a PonderTalk operation and forward the invocation to the **task** object.
	External interface of the role	None.
All exposed notifications	Mission-specific role	Add code that registers these notifications in the $exposedNotifcations$ table of the $Role$ class.
	External interface of the role	None.
Required notifications	Mission-specific role	Add code that registers these notifications in the $requiredNotifcations$ table of the $Role$ class.
	External interface of the role	None.

Table 7.1: Generating Role Code from the Specification

and their corresponding state. These methods are used by the $StateAggregator$ class, which implements the *State Aggregator* entity.

The *State Aggregator* is responsible for collecting state information from the managed

roles and providing the information to the *Management Tree Node* entity as well as aggregating and sending state updates to the manager role. It is also responsible for generating the appropriate notifications when events such as communication link disconnection and permanent UAV or link failure occur and updating the waiting list accordingly using the **addToWaitingRolesList (RoleIdentity roleId, State state)** method of the role (Section 7.3). It does so when the state information is not received within the specified timeouts.

The *State* data structure is used to communicate and persist the state of a role. The management and operational state of the role is aggregated and stored by persisting the *Management Tree Node* instance of the role and operational states that are dependent on the specific role. The management state mainly contains the list of role identity objects of the active (non-failed) roles managed by this role. The operational state is a list of name-value pairs added by the role that is sending the state update. The role adds the operational state to the table maintained by the *StateAggregator* using the *addOpState* method provided by the *StateAggregator* class.

```
addOpState(String name, Object object)
```

Unlike the management state, which is interpreted by the framework, the operational state is passed to the role receiving the state update as it is, and the role has to understand and interpret the semantics of the state. State update from a role could be (1) a default *keep alive* message (consisting of role identity number, location, speed and direction) in which case the *State* data structure is not used for efficiency reasons, or (2) management state only, in which case, the management tree node is persisted and added to the state, or (3) operational state only, in which case, a copy of the operational state table maintained by the *State Aggregator* is added to the state, or (4) complete state, i.e., both management and operational state. The *StateAggregator* class provides the *setStateUpdateType* method to allow adaptation of state update types through policies. The type can be one of the strings – *alive, management, operational, complete*. Roles implement the *setStateUpdateType* : PonderTalk message and forward the call to the *Role* class, which invokes the method provided by the *StateAggregator* class.

```
@Ponder2op("setStateUpdateType:")
setStateUpdateType(String type)
```

Manager roles' policies set the state update type of their managed roles (e.g., Figure 6.17, line 59). While the choice is mission dependent, setting the update type of managed roles remotely is less conflict prone than using local policies to set the update types at the managed roles themselves since this would create difficulty in aggregation at the manager role when differing update types are used by managed roles. Manager roles, however, set their own state update type (what they send to their managed roles) using local policies.

On manager roles the $StateAggregator$ class performs bundling of state update messages received from managed roles so as to decrease the communication protocol overheads and best utilise the communication bandwidth. When the state update type is *alive* roles send their identity number, location and velocity to their manager. This information is bundled in a flattened form of – id_1, x_1, y_1, $speed_1$, $theta_1$, id_2, x_2, y_2, $speed_2$, $theta_2$, ..., id_n, x_n, y_n, $speed_n$, $theta_n$ where n is the number of roles managed by the manager role. If the mission solely relies on the rendezvous algorithm (i.e., the movement-based link maintenance approach is turned off), aggregation can be performed at each manager role by computing average locations and summation of velocity components for resultant direction estimation in order to decrease the size of the state update message. When the state update type is *management*, roles persist and send their management tree nodes. From these data structures, the $StateAggregator$ extracts the role identity object of children roles, if any, bundles them into a list, adds the list to the management tree node, persists and sends the management tree node. Note that while the role identity number is a unique number used to identify roles, as discussed before, the role identity object is a complex data structure containing, among other data, information that enables collaborating roles to import remote role references. While the role assignment and confirmation, and default *alive* messages suffice for formation and maintenance of hierarchical domain structures respectively, for other structures, additional messages are required for formation and maintenance. Hence, when a mission needs a domain structure different from a hierarchical one, it has to set the state update type to *management* or *complete* to allow the role references to be disseminated beyond their manager and managed roles.

Discovery and Optimisation

The *Discovery* entity periodically broadcasts discovery beacons over UDP. However, subsequent communication between this entity and the discovered UAV uses the secure communication channel implemented by the *Communication Handler* entity in the communication layer. The *Discovery* entity, implemented by the $UxvDiscovery$ class, provides several PonderTalk operations.

The $has:assign:$ operation, which is used by role assignment policies, is implemented by the $assign$ method.

```
@Ponder2op("has:assign:")
assign(P2Array requirement, String roleType)
```

This method adds the role type and its required capabilities to the list of roles and their capability requirements maintained by the $UxvDiscovery$ class. The list uses the $BloomFilter$ data structure to represent the capability requirements.

When a UAV is discovered, the $adduav:cap:$ PonderTalk message, implemented by the $addUav$ method of the $UxvDiscovery$ class, is invoked by policies.

```
@Ponder2op("adduav:cap:")
addUav(String uav, String capability)
```

If the UAV satisfies the capability requirement of at least one role in the role capability requirements list, it is added to the $BipartiteGraph$ data structure built by the $UxvDiscovery$ where the roles are one bipartition and the UAVs are the other and the edge between them is the aggregate utility associated with assigning the role to the UAV. As discussed in Chapter 4, the aggregate utility is a measure of the benefit of assigning a role to a UAV considering the benefit of assigning that role, assigning it to the specific UAV and using the specific UAV. It is a weighted sum of role utilities, role-system utilities and system utilities with weights that are set through policies (discussed in Chapter 4). The $AggregateUtility$ class provides the $utility:setweight:$ operation, which is used by policies to set weights for utilities.

```
@Ponder2op("utility:setweight:")
setWeight(String name, String weight)
```

In the $BipartiteGraph$ data structure, both roles and UAVs are represented in terms of their capability requirements and capability provisions respectively. The capability sets are represented using Bloom filters [Blo70] in order to enable efficient storage and comparison of capabilities. Our implementation of bloom filters uses the lower bound for false positives and the optimal value for the number of hash functions as described in [BM04]. The parameters of the Bloom filter including the maximum allowable false positive and the type of hash function can be changed through policies. The MD5 digest is used as a default hash function. The $BloomFilter$ data structure provides methods for performing set operations. For example, whether a UAV satisfies the capability requirement of a role is determined by using the **isSubSet** method.

```
isSubSet(BloomFilter a, BloomFilter b)
```

The broadcast rate and other behaviours of the discovery service can be adapted through policies. For example, the discovery service may trigger the *Optimiser* when as many UAVs are discovered as are required to fulfil available roles if this behaviour is enabled through the $opportunisticOpt$: operation (discussed in Section 6.8, for policies shown in Figure 6.16).

```
@Ponder2op("opportunisticOpt:")
opportunistic(boolean flag)
```

The *Optimiser* entity fetches or receives the bipartite graph depending on the policies managing the behaviour of the *Discovery* and *Optimiser* services.

The *Optimiser* entity is implemented by the $Optimiser$ class. It computes a minimum cost maximum matching of the bipartite graph, using the $O(n^3)$ improved Hungarian algorithm [Kuh55, Kuh56, Mun57, BL71, CMT88], and provides the result to the role, which uses it to perform the role assignments. The rate of optimisation, until all available roles managed by the role performing the assignment are assigned, can be set by the **setOptimRate** method provided by the $Optimiser$ class. In previous examples, this parameter is also referred to as the waiting period before optimisation since it controls the period between the discovery of the first UAV and the beginning of the optimisation process.

```
@Ponder2op("setoptimrate:")
setOptimRate(int rate)
```

7.4.1 Capability

The capability of a UAV is the set of operations that its software and hardware support as well as the events it generates. This depends on the current set of software tasks loaded on to the UAV. The capability specification of a UAV is generated dynamically by querying the tasks in a UAV. The *Capability Manager* entity is responsible for querying and preparing the description. It is implemented by the *Capability* and *CapabilityAdvertiser* classes, which generate and advertise the description respectively. As stated before, it is assumed that tasks can be queried for their interface description. A task implements a task interface with a naming scheme where the interface name is the task name suffixed by an 'I'. For example, an *Explorer* task implements an interface called *ExplorerI*. To facilitate the capability description generation, task interfaces are annotated using two annotations, namely *TaskInterface* and *TaskEvent*. The *TaskInterface* annotation is used to mark (and indicate the name of the corresponding task) that an interface is a task-interface, which implies that it has operations/notifications that can be included in the capability description. This marking is used later, while generating the capability description, to differentiate between the various interfaces a task implements. The *TaskEvent* annotation is used to mark events so that it would be possible to differentiate between the task's operations and its events. Our algorithm for capability description generation of a single task is shown in Algorithm 4. The algorithm reads the interfaces implemented by the task, using Java Reflection, and decides whether to consider the interface in general or the methods of the interface in particular by using the annotations. The *Capability* class provides methods to query for low-level capability description (**generateLowLevelCapabilityDesc**), full capability description (**generateCapabilityDesc**) and check support for specific capabilities (**haslowcap:**).

```
@Ponder2op("genlowcapdesc")
generateLowLevelCapabilityDesc()
@Ponder2op("gencapdesc")
generateCapabilityDesc()
@Ponder2op("haslowcap:")
hasLowLevelCap(String cap)
```

When a UAV is attempting to join a team, these methods are used by the class to query for the capability descriptions of the UAV and send it to the remote role performing the discovery. If the attempt succeeds and the UAV is assigned to a role, the remote role concludes its communication with the $CapabilityAdvertiser$ by sending a role assignment message that the $CapabilityAdvertiser$ uses to invoke the **loadrole** method of the $RoleManager$.

Algorithm 4 Algorithm for Capability Description Generation

```
Determine all interfaces implemented by the task.
for all interfaces implemented by the task do
  Check for a @TaskInterface annotation.
  if the annotation is detected then
    Check if the task matches the task indicated in the annotation.
    if it matches then
      for all methods in the interface do
        Check for a @TaskEvent annotation.
        if the annotation is detected then
          Add the method to the description as an event.
        else
          Read the argument types.
          Add the method as an operation.
        end if
      end for
    else
      return Error
    end if
  else
    return Empty description
  end if
end for
return Description
```

7.5 Communication Layer

The communication layer consists of the *Message Sender*, *Message Router*, *Communication Handler* and *Communication Link Maintenance* entities.

The *Message Sender* entity is implemented by the $MessageSender$ class, which provides a message passing method that is used by framework elements that need reliable message sending. Framework elements can create their own message sender object with the required timeout and retry limits.

The *Message Sender* entity uses the *Communication Handler* entity for secure communication. The *Communication Handler*, implemented by the $Communication$ class,

is built upon UDP and provides a secure communication channel between UAVs. As described in Chapter 4, the Certificate Public Key Infrastructure (C-PKI) system is used to exchange a common secret key generated using the Diffie-Hellman protocol between each manager and managed role. The communication class provides the $createSecure : key :$ operation that is used to create a secure channel during the management framework's bootstrap stage by using the certificate ($certFile$) and public key ($keyFile$).

```
@Ponder2op("createSecure:key:")
Communication(P2Object myP2Object, String certFile, String keyFile)
```

The extra argument, $myP2Object$, is not part of the $createSecure : key :$ PonderTalk message. It is a mechanism used in the Ponder2 implementation to get a reference to an adaptor object that represents the managed object instance created by constructors (note that the above method is a constructor for the $Communication$ class).

The *Message Router* entity, implemented by the $MessageRouter$ class, handles incoming messages for multiple roles (and other framework entities) residing in a UAV. This entity enables registration of other entities to receive packets of a certain type and/or source as well as de-registration. Entities register using the $register$ method of the $MessageRouter$ class by indicating the type of messages they are interested on and providing their reference using the $MessageRouterRegistration$ data structure.

```
register(MessageRouterRegistration reg, MessageReceiver receiver)
```

The $MessageRouterRegistration$ data structure consists of the message type and source information. All entities that receive messages from other UAVs implement the $MessageReceiver$ interface.

In the case when an entity registers to receive packets of more than one type (or source) that intersect, the registrations are aggregated. A separate exclusion table is kept when an entity de-registers, if that entity and the registration it is de-registering from lies in an aggregate registration. When a new packet arrives, the dispatch table as well as the exclusion table are checked before the packet is passed to the registered roles.

The *Communication Link Maintenance* entity, which deals with the two communication link maintenance algorithms, was implemented by my colleague Dr. Anandha

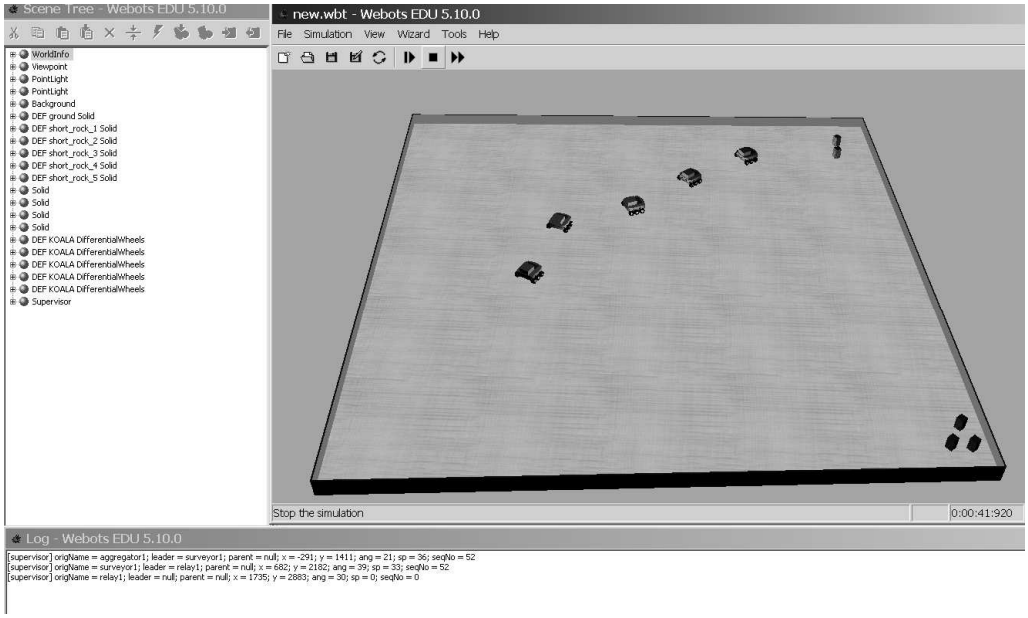


Figure 7.3: Snapshot of Webots Simulation

Gopalan. This entity was implemented and evaluated in the Webots [Ltd] simulator but was not tested on actual robots due to difficulty in controlling the wireless network range. A snapshot of the simulation is shown in Figure 7.3 and simulation-based evaluation of the algorithms is presented in Chapter 8.

7.6 Proof-of-concept Demonstration

The self-management architecture was tested on the Koala robots [kt]. The Koala robot is a mobile robot that has 16 infra red proximity sensors around the body of the robot, and a camera. It is controlled by an Asus EEE PC[3] running windows and Java through a USB to serial cable. The scenario chosen for the demonstration was a search and rescue mission of a wounded soldier. The soldier is assumed to possess a wearable computer and a body sensor network that monitors the soldier's condition. The wearable computer was another Asus EEE PC, while the commander was a laptop and two robots were used as the unmanned vehicles. None of the wireless devices were connected to the infrastructure and all the devices were part of the same ad hoc network. The steps are as follows:

(i) Soldier is wounded in the battlefield,

[3]Intel(R) Celeron(R) M CPU 900 MHz, 512 MB RAM.

(ii) Wearable computer sends a distress signal to the base reporting on the soldier's condition,

Figure 7.4: Snapshot of Proof-of-concept Demonstration – Distress

(iii) The *Commander* assembles the mission for assistance, comprising unmanned vehicles capable of navigation, communication, surveillance and hazard detection. In this scenario, two roles are assigned: the *Surveyor* and the *Aggregator*,

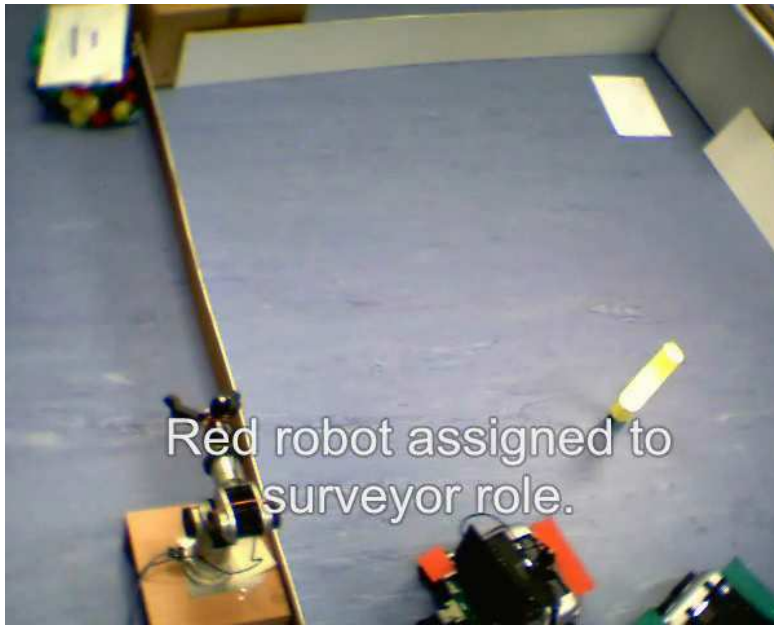


Figure 7.5: Snapshot of Proof-of-concept Demonstration – Mission Assembly

(iv) The *Surveyor* starts to move towards the soldier and detects a hazard along the

way,

Figure 7.6: Snapshot of Proof-of-concept Demonstration – Hazard

(v) On detecting the failure, the *Surveyor* role is reassigned to the *Aggregator* by the *Commander*. Since the *Commander* has been collecting state information through the management tree, the last position of the previous *Surveyor* is estimated from the last received state update so that the new *Surveyor* can avoid the "hazard",

(vi) The new *Surveyor* is able to avoid the hazard using the information provided,

Figure 7.7: Snapshot of Proof-of-concept Demonstration – Hazard Avoided

(vii) The new *Surveyor* reaches the soldier and delivers assistance, as necessary. This proof-of-concept demonstration was shown as part of the Annual Conference of the ITA, 2008 (ACITA 2008) [ACI]. Snapshots of the demonstration are shown in Figures 7.4 - 7.8. The first *Surveyor* robot stops on detecting the "hazard" (which is a yellow cylinder). The second *Surveyor* (which used to be the *Aggregator* before the role reassignment) avoids the hazard and reaches the soldier.

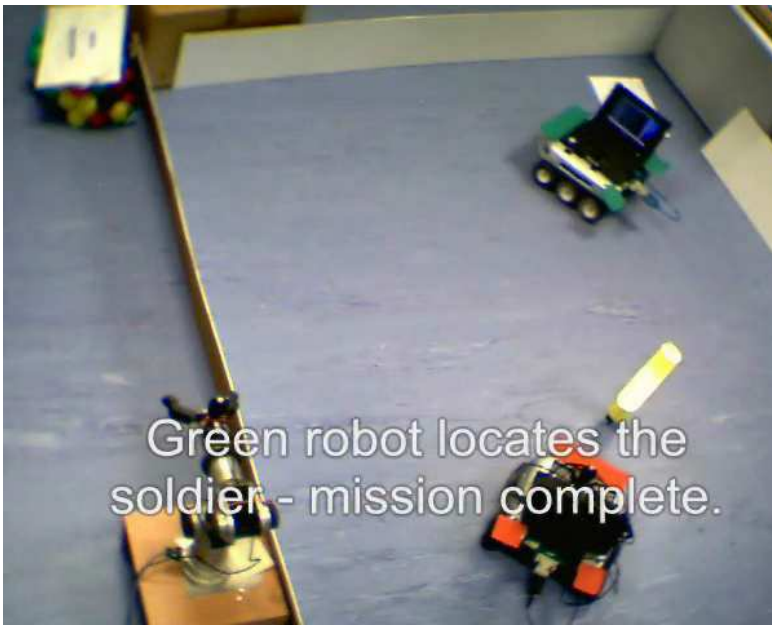


Figure 7.8: Snapshot of Proof-of-concept Demonstration – Mission Completed

7.7 Summary

In this chapter, we have presented the prototype implementation of the self-management framework, which was built upon the self-managed cell. The implementation is modular in that one can seamlessly add or remove entities from any of the three layers. The prototype implementation was tested on the Koala mobile robots.

Chapter 8

Evaluation

The self-management framework enables the specification of missions in terms of roles whose behaviour is controlled through policies. In accordance with this specification, a self-managing team is formed and maintained. While these approaches were demonstrated in the case study chapter, in order to determine the applicability of the framework in real-life missions, we evaluate its performance using three broad measures – response time, success rate and communication overhead. The first measure is crucial since the framework is targeted for time critical missions such as disaster management and military applications. The second measure is relevant to some behaviours of the framework such as optimal role assignment and communication link maintenance. The third one is also important since communication resources are scarce in wireless applications in general and in mission environments targeted by the framework in particular.

In this chapter, we evaluate our self-management framework through analysis, and performance evaluation of the prototype implementation. First, we study the communication overhead incurred by formation and maintenance of the management tree. The communication overhead is evaluated analytically. In doing so, in addition to the overhead of forming and maintaining the hierarchical management structure, we are able to easily evaluate the overhead that would be introduced for different collaboration structures since the management tree is used to maintain the domain structure, which is used for collaboration purposes. We then present results of experimental evaluation of the prototype implementation performed to evaluate the performance

of the framework with respect to response time (e.g., mission setup and failure response) and success rate (e.g., communication link maintenance) and conclude with a critical analysis of the framework.

As mentioned in Chapter 4, the hierarchical management structure, while having several advantages, introduces an inherent latency in mission setup time due to the fact that a manager role itself has to be discovered and assigned to a role before it performs assignment (i.e., propagation delay introduced by the hierarchy). Since the time it takes to form the team is crucial in target applications of the self-management framework such as disaster management, we ask whether the hierarchical structure introduces too much of a delay that could have been avoided by using another viable structure. We hypothesise that although the performance of the hierarchical approach for management with respect to mission setup time is largely affected by the delay introduced by the hierarchies, the distribution of management decisions would result in a significant gain in time to enable the hierarchical structure to outperform the centralised structure for higher number of roles.

In Section 8.2.1, we evaluate the above hypothesis by comparing the hierarchical structure to a centralised structure where there is no communication delay in propagation since there is only a single manager role (i.e., the commander). We also study the effect of distribution of management responsibilities for smaller and larger size teams in Section 8.2.2.

We do not consider other structures such as peer-to-peer for comparison since these structures do not capture the management semantics and hence are not viable options as a management structure. However, since the collaboration organisation structure maintained by the domain structure can have an arbitrary organisation structure, we do not rule out other structures in the message complexity analysis of the management tree maintenance, since the tree can be used to form and maintain these structures.

The self-management framework enables adaptation to failure that is achieved by means of policies as illustrated in the case study (Chapter 6). The response time to failure is a crucial parameter and is controlled by the failure timeouts. As long as the mission administrator has chosen the failure timeouts depending on the required responsiveness of the mission to failed roles and specified policies to manage failures

of different roles, the framework will respond to failures of UAVs after the specified timeouts. However, we ask, what if the failure involves too many roles simultaneously – how would the framework perform? This is necessary in real-life missions where UAVs in geographic proximity could fail simultaneously due to the nature of the environment, or attacks in military applications. The evaluation presented in Section 8.2.3 is performed to answer this question.

In Chapter 4, we have argued that assigning UAVs as soon as they are discovered, which we referred to as immediate role assignment, may lead to an incomplete team while there are sufficient UAVs to fulfil all the roles of the team. The evaluation presented in Section 8.2.5 is performed to validate this argument by using a sample set of UAVs and roles and random arrival sequences of UAVs and showing that when using the optimised assignment approach, the framework achieves a complete team in all cases as opposed to the 80% success rate achieved when using the immediate assignment approach. We also study the time taken to perform assignment by each approach.

The communication layer tries to maintain communication links among UAVs by using two approaches – one based on movement control and the other based on rendezvous setup. The success of these two approaches is measured in different ways. While for the first approach success is measured by whether the team is able to move in a convoy fashion thereby maintaining communication links, for the second one the measure is whether the members of the team manage to arrive at the rendezvous point in time. Note that because UAVs know the rendezvous location whether or not a UAV will reach at the point is not the concern since as long as its motion capability is not impaired, a UAV will eventually reach there unless it fails (UAV failure) in which case the issue is the concern of the failure management protocol.

For the first approach, the issue concerns the conditions at which the team would succeed in moving in a convoy fashion. In Chapter 5, we have stated that the choice of the range threshold and state update rate have considerable impact on the success rate of the approach. We have also argued, in Chapter 4, that a large range threshold and infrequent state update rate results in lower performance with respect to success rate. The evaluation presented in Section 8.2.6 tests this hypothesis. For the second approach, since time of arrival instead of arrival is the issue, we test whether the

algorithm's estimate of speed (the average speed) results in arrival of all the UAVs within the specified time. We also investigate the performance of the algorithm with respect to the success rate (arriving at the rendezvous point within the specified time) when each UAV continues to the rendezvous point at its current speed and top speed.

8.1 Message Complexity

8.1.1 Model

For the purpose of message complexity analysis, we represent the UAV wireless network using unit disk graphs [CCJ90], which are widely used as an idealised representation of wireless networks for analysis. In this model, nodes are assumed to be placed in an Euclidean plane and have identical communication range of one unit (C_R normalised to one). Two nodes are then connected by an undirected edge (bidirectional communication link) if the distance between them is at most one, i.e., both nodes lie within the intersection of the two unit radius circles drawn around each of the nodes.

Hence, we represent the UAV ad hoc network using the resulting graph $G_A = (A_{UAV}, L)$ where A_{UAV} is the set of available UAVs and L is the set of bidirectional communication links (edges) between the UAVs in the unit disk graph. Each link is denoted by l_{ij} where UAV_i and UAV_j are the vertices (UAVs) connected by the link in the unit disk graph. An example set of UAVs and the corresponding unit disk graph is shown in Figure 8.1.

We make the following assumptions:

1. The communication range of all UAVs is identical. From this assumption, it follows that all links are bidirectional, i.e., we consider two UAVs to be linked only when they are within the transmission range of each other.

2. At the time a manager role is broadcasting its discovery beacon, there are at least as many UAVs as needed by this role. The tree formation algorithm does not make this assumption; as the discovery service broadcasts periodically it will eventually discover as many UAVs as needed. However, this would make

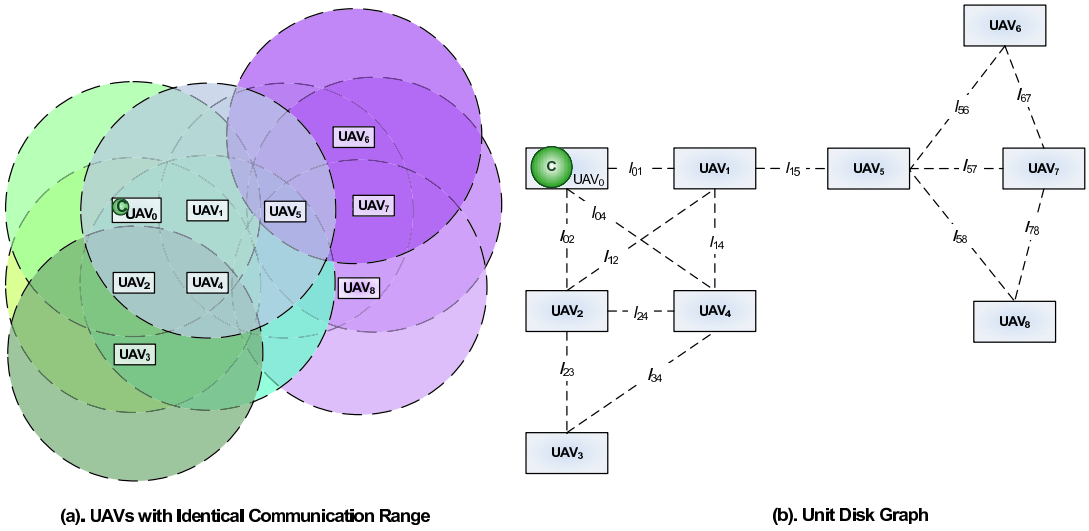


Figure 8.1: UAV Network Model

the message complexity unbounded or at least dependent on the time it takes for UAVs to be available. We will perform the analysis disregarding the fact that UAVs may not be available from the start but discuss the implications of this and the other assumptions later on.

3. Local broadcast [Per98], i.e., the delivery of data to every UAV within range of the transmitter, is supported by the UAVs wireless network interface (e.g., it is supported if UAVs use IEEE 802.11b). This, with the previous assumption (assumption (2)), leads to a single discovery message per manager role.

4. UAVs do not move out of the direct (single-hop) communication range, with respect to their manager UAV, while the discovery process is taking place.

5. The size of messages is independent of the size of the team (number of roles) and each message generated by the self-management framework incurs only one message at the network level.

8.1.2 Message Complexity of the Management Tree Formation

The management tree is formed through a series of discovery broadcasts made by each manager role in the mission specification. In doing so, a spanning tree $G = (V, E)$ where V is the set of roles and E is the set of edges is formed. The first UAV (UAV_0 in Figure 8.1) to perform discovery is the one enacting the *Commander* role. The number of messages sent over the link between the discovering UAV and the one

that is being discovered depends on the degree of success of the discovery. Assuming that all UAVs have the necessary credentials and hence will pass the authentication stage, there are three possible values for the number of messages sent over the link between the discovering and discovered UAV as shown below.

Link between the discovering and discovered UAV	**Number of Messages**	**Remark**
Any link	11	1+2*5, i.e., 1 join request message, 2 signed certificate-exchange messages and 2 Diffie-Hellman exchange messages, 1 capability summary message
Link to potential team-member UAV	15	11+2*2, i.e., 11 messages + 1 capability request and 1 capability description messages
Link to UAV assigned to a role	19	15+2*2, i.e., 15 messages + 1 role assignment message + 1 role confirmation message

Except for the discovery and join request messages, all other messages, including the role confirmation message, are acknowledged and hence each incur one more message. This could, to some extent, be optimised by piggybacking acknowledgement with a message in the other direction. The role confirmation message is not a simple confirmation since it consists of information used by the parent role to import the assigned role. Hence, it is an acknowledged message as the others.

Now assume the mission class specification of the case study (Chapter 6) with the corresponding mission instance specification (modified to contain a single *Medic* and two *Transporter* roles for the sake of space) are used. Also, assume the capabilities of UAV_1 & UAV_4 are suitable for the *Aggregator* and *Relay* roles respectively. As shown in Figure 8.2 (a), the *Commander* discovers UAVs 1, 2 & 4 and assigns the *Aggregator* and *Relay* roles to UAVs 1 & 4 respectively.

In Figure 8.2, the edges labelled as e_{xy} correspond to the management tree edges formed as a result of this assignment where x and y are roles. For example, e_{CA} is the edge between the *Commander* role and the *Aggregator* role. The number of messages sent by the *Commander* UAV for performing this discovery and assignment is 1 discovery message, 19 messages through l_{01}, 15 messages through l_{02} and 19 messages through l_{04}.

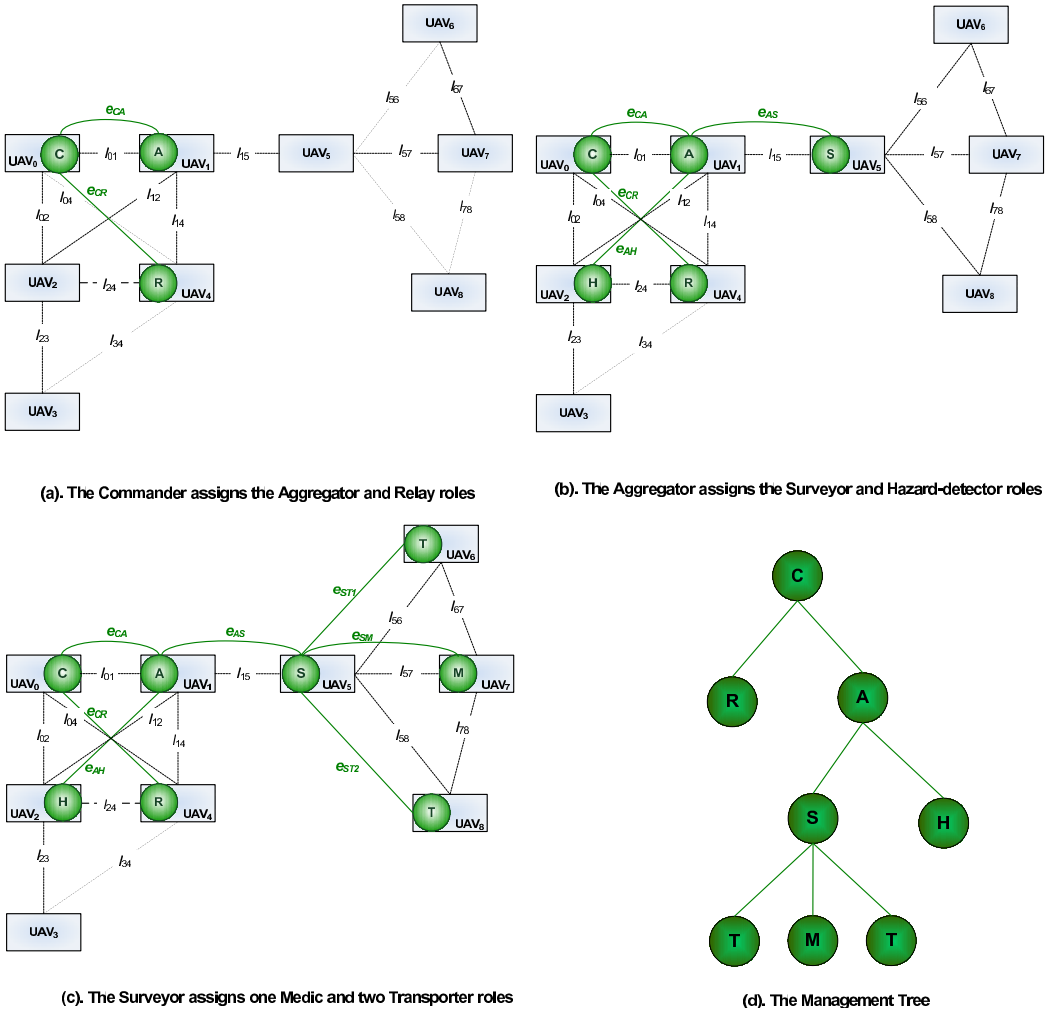


Figure 8.2: Management Tree Formation

The discovery and assignment process is in turn repeated by the assigned roles until all roles are assigned. In Figure 8.2 (b), the *Aggregator* discovers UAVs 5 & 2 and assigns the *Surveyor* and *Hazard-detector* roles to them respectively. In Figure 8.2 (c), the *Surveyor* discovers UAVs 7, 6 & 8 and assigns one *Medic* and two *Transporter* roles to them respectively. The resulting management tree is shown in Figure 8.2 (d).

Each of these assignments incur 19 messages resulting in a total of 7*19 messages incurred by all discovered and assigned UAVs as there are 7 roles (excluding the *Commander*). In addition, there is one discovery message associated with each manager role resulting in a total of 3 discovery messages sent by all manager UAVs as there are three manager roles (*Commander*, *Aggregator*, *Surveyor*). Also, 15 messages were incurred when the *Commander* discovered UAV_2 although it did not assign it to a role. The total messages, sent by all involved UAVs, over the network in forming the

example management tree is then: 3 + 7*19 + 15. Now let us generalise this result to estimate the message complexity of the management tree formation.

- In our example, there were 3 discovery messages corresponding to three manager roles out of a total of 8 roles in the mission. In general, given $n = |V|$ roles, there can be as many as $n-1$ manager roles (forming a degenerate tree/a chain) resulting in $n-1$ discovery messages.
- All the successful assignments result in an edge in the management tree G. As there are $|V|-1$ edges in G there will be a total of $19(|V|-1)$, i.e., $19(n-1)$ messages incurred by all successful (assigned) UAVs.
- Unsuccessful UAVs, i.e., UAVs that are authenticated but not assigned as not needed because of unsuitable or less suitable (lower utility) capability, incur either 11 or 15 messages depending on whether they were candidates (lower utility) for assignment or not (unsuitable). In our example, only UAV_2 incurred these messages as it was the only one within the direct reach of two manager roles. In general, all the UAVs that would later be assigned to one of the roles, i.e., $n-1$ UAVs, could be within the direct reach of each other. As mentioned before, in general, there can be as many as $n-1$ UAVs (forming a chain) resulting in $n-2$ (i.e., $n-1$ less the assigned UAV) unsuccessful UAVs for each assignment. Taking the higher number of messages, i.e., 15, the total number of messages incurred by these unsuccessful UAVs could be as much as:
  $15 \sum_{i=1}^{n-2} i = \frac{15}{2}(n-2)(n-1)$

Hence, to form a management tree $G = (V, E)$ with $n = |V|$ nodes (roles) and $|E| = |V|-1$ edges, the (maximum) total number of messages sent by all involved UAVs over the network is:

$$(n-1) + 19(n-1) + \tfrac{15}{2}(n-2)(n-1) = \tfrac{5}{2}(3n^2 - n - 2)$$

Consequently, the message complexity for the management tree formation is $O(n^2)$.

There are three points worth noting: (1) the overhead caused by member UAVs irrespective of the team size (number of roles) is only 20 or 19; since the total number of messages sent by a successful UAV (team member) is either 20 or 19 depending

on whether it is a manager role or not respectively, (2) to study the worst case complexity, we have considered an unrealistic team structure such as a management hierarchy where every role except one is a manager role thereby resulting in a degenerate management tree (a chain). It is worth noting that the polynomial degree complexity comes from this scenario. In reality, a mission administrator would not specify a mission class that would result in this management tree and hence the average case complexity can be significantly better, (3) on the other hand, there is one possibility we have not considered in the analysis – in general, there may be n' number of unsuccessful UAVs within the direct communication range of the discovering UAV. This number, n', is limited by the communication range of the discovering UAV. This would raise a scenario where the message complexity that would be $O(n * n')$ is unbounded with respect to n, i.e., number of roles in the team and be of polynomial complexity (if $n' = c*n$ where c is a constant) even with the absence of the hypothetical degenerate-tree management hierarchy.

Going back to the implications of our assumptions stated in Section 8.1.1, assumption (1), i.e., all UAVs have identical communication ranges, does not hold true in a team composed of heterogeneous UAVs. The difference in communication range results in some links being unidirectional thereby leading to the possibility that some UAVs in the transmission range of the discovering UAV not being discovered. However, this does not impact the actual message complexity as long as local broadcast (assumption (3)) is used by the discovering UAV, since one message is sent for discovery irrespective of the number of receivers.

Assumption (2) that all required UAVs are within range at the start of the discovery process may not be true. This would require periodic rebroadcast of the discovery message. Consequently, the communication overhead introduced by the discovery messages will be higher than the value used in the analysis (a single broadcast/discovery message for each manager role). As the manager role broadcasts the discovery message periodically, the overhead depends on the mission's broadcast rate (the $discoveryRate$ mission parameter) and UAV availability.

Assumption (4) and assumption (5), i.e., that a managed UAV not moving out from the direct communication range of the manager UAV for the duration of assignment and message size being independent of the number of roles respectively have more significant implications on the communication overhead of maintenance of the man-

agement tree than on the formation and are discussed in the next section.

8.1.3 Message Complexity of the Management Tree Maintenance

The management tree, which is formed as a result of a series of discovery and role assignments, is maintained by means of state aggregation and dissemination. Each role (node) uses the management tree, which is a spanning tree, to propagate its state. Consequently, the total number of messages sent over the network for one round of state update is twice the number of edges of the tree, i.e., $2(|E|) = 2(|V|-1) = 2(n-1)$, as each role sends one message and receives another message on each state update. Hence, the message complexity for one round of state update is $O(n)$ and the communication overhead for maintaining the management tree is $2(n-1)/\tau$ messages per second where τ is the state update rate (corresponding to the $stateUpdateRate$ mission parameter) in seconds. This result is based on the assumption that the size of the state update messages is independent of the size of the team (number of roles). However, since the state update messages contain (1) location and velocity of UAVs, and may include (2) information used for domain structure (collaboration organisation structure) maintenance, the size of the message is dependent on the size of the team. Since manager nodes aggregate state updates, the first element (location and velocity) can be safely assumed to have a constant size for moderately sized teams (e.g., 50 UAVs) or a multiple of a constant size for larger size teams. This is because the size of the state information included in the state update message per UAV is small enough that a single state update message can contain state information of a number of UAVs. The location components (two dimensions), speed and direction are each represented in 4 bytes float data type accounting to a total of 16 bytes per UAV and the role identity number (not object) consists of the mission ID (1 byte), hierarchy level (1 byte) and sequence number ($d-1$ bytes where d is the number of hierarchy levels in the team). For example, for a team consisting of 50 UAVs with 5 hierarchy levels a total of 22 bytes (16+1+1+4 = 22) per UAV is used for the state update and hence the aggregated state will have (n-1)*22 = 49*22 = 1078 bytes which becomes 1129 bytes, after inclusion of the framework (23 bytes) and UDP/IP headers (20 bytes + 8 bytes), is well below the MSDU (MAC service data unit) size (2304 bytes) [IEE07] of an 802.11 system (assuming UAVs are using this system).

As mentioned in Chapter 4, the collaboration organisational structure (maintained in the domain structure) is mission dependent. Manager roles get the information that enables them to import remote role references to their managed roles from the role confirmation messages and managed roles get similar information pertaining to their manager roles from the role assignment message. Hence, neither the formation nor the maintenance of a hierarchical domain structure, which is sufficient for missions whose role interactions are constrained between manager and directly managed roles (e.g., the *Surveyor* role and its managed *Medic* and *Transporter* roles in the case study), does need additional communication. The mission administrator only needs to specify policies to import role references during assignment and the domain structure is formed (e.g., the *Surveyor* role policies for importing *Medic* and *Transporter* role references shown in Figure 6.19, lines 67 - 71). The domain structure maintenance does not need additional information in the state update message as a manager role will be able to know the status of its managed roles from the default *keep alive* message.

However, as mentioned in Chapter 4, in general, missions may have roles whose interactions may go as far as involving all roles, and as mentioned in Chapter 7, collaboration organisation structures different from the hierarchical one require a *management* state update type that involves aggregation and sending of role identity objects. Although role identity objects are also aggregated by manager roles, the same assumption (single message per update) cannot be made since these are complex objects with significant size. Recall that role identities are data structures consisting of role identity number, the role name (type), last known location and velocity, the remote address and path of the role.

In the following, we investigate the communication overhead of the management tree maintenance when a complete domain structure (peer-to-peer collaboration organisation structure) and a partial domain structure are used by the mission to study the overhead in the most extreme and likely role interaction patterns respectively.

Maintaining a Complete Domain Structure on All Roles

The size of the messages involved in the domain structure maintenance largely depends on the number of roles as can be seen in Figure 8.3. This makes the com-

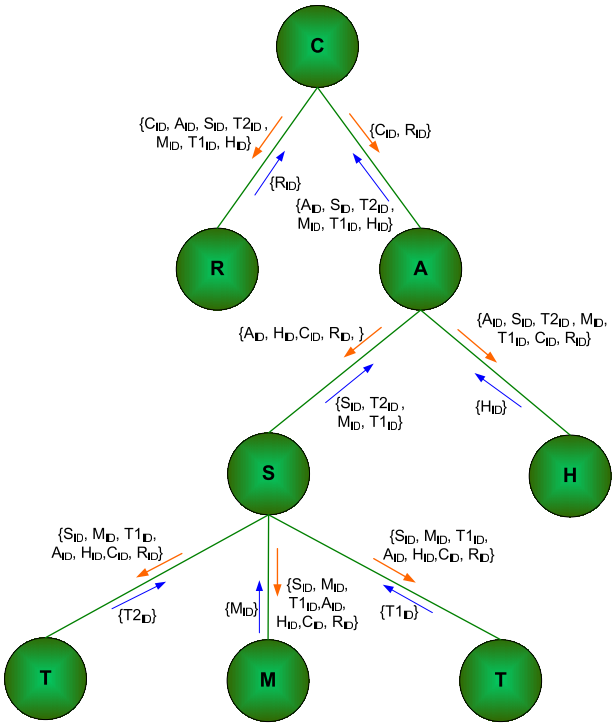


Figure 8.3: Management Tree Maintenance with Complete Domain Structure

munication overhead dependent on whether all roles in the team need to be aware of each other or not.

Figure 8.3 shows the state updates that are sent up and down the management tree (location and velocity not shown) when it is the case that all roles should be aware of each other, i.e., the collaboration organisation structure is peer-to-peer.

Each role sends its identity, directly managed role identities and indirectly managed role identities up the management tree and receives identities of all other roles except its own and its (directly and indirectly) managed roles. For example, the message from A (the *Aggregator* role) to S (the *Surveyor* role) contains the *Aggregator* and *Hazard-detector* role identities since the *Aggregator* manages the *Hazard-detector* role. This is necessary when all roles require an operation and/or notification from each other since it allows them to update the domain structure when UAVs join and leave the team. Recall from Chapter 7 that the role identity object contains the necessary information to import a reference of a remote role into the domain structure. On each edge of the management tree, the sum of the number of role identity objects included in the state update messages (sent and received) is equal to $|V|$, i.e., the number of roles. Hence, the communication overhead for maintaining the domain

structure is $|E| * |V|/\tau = (n-1)n/\tau$ role identity objects per second where τ is the state update rate in seconds.

Maintaining a Partial Domain Structure

Recalling the search and rescue mission considered in Chapter 6, to which the example management tree that we are using in the analysis refers, all roles did not need to interact with each other. In addition, the interactions involve operation invocation by manager roles on their managed roles but not vice versa. The communication overhead can be decreased for such types of missions by limiting the scope of the information contained in the downward state update. A manager role can send information pertaining to the sub-team consisting of itself and its (immediate/directly managed) children as shown in Figure 8.4. This results in each role maintaining a partial domain structure that comprises its immediate children, its manager and sibling roles if the role itself is a manager role (e.g., the *Surveyor*). If it is not a manager role (e.g., the *Medic* and the *Transporter*s), it only needs to maintain manager and sibling roles. In addition, since the upward state update message sent and received by a manager role contains information about the indirectly managed roles, each manager role can add its indirectly managed children to its partial domain structure resulting in a partial domain structure with nested hierarchical organisation shown by the shaded concentric regions in Figure 8.4.

The communication overhead for maintaining the management tree is then $|E|k/\tau = (n-1)k/\tau \leq (n-1)n/\tau$ role identity objects per second where τ is the state update rate in seconds and $k \leq n$ is the average number of messages (sent and received) per edge. In the example tree shown in Figure 8.4, $n = 8$ and $k = 32/(n-1) = 32/7$ since a total of 32 role identity role objects are sent and received over all edges. Compared to the 56 messages $n(n-1) = 8*7$ sent over all edges for maintaining a complete domain structure, the partial domain structure maintenance reduces the communication overhead by more than 40%. The value of k depends on the mission specification (management hierarchy as well as actual number of instances of roles of each type) but is always less than n unless the management hierarchy is a centralised one, i.e., all roles are directly managed by the *Commander* role. The mission administrator can enforce this behaviour of partial domain structure maintenance using policies

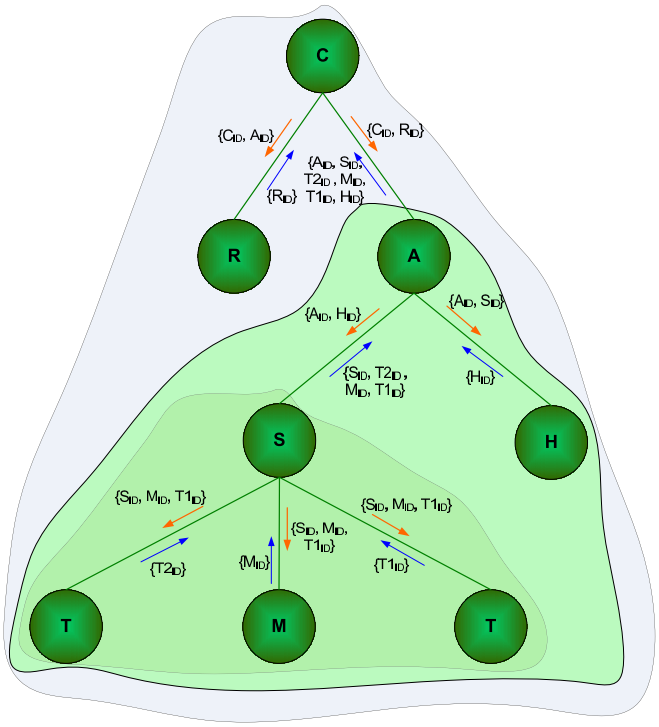


Figure 8.4: Management Tree Maintenance with Partial Domain Structure

(discussed in Chapter 7) depending on the interaction of roles. Other forms of partial domain structure maintenance such as hierarchical coalitions can also be used to control the communication overhead.

In general, state update messages contain both management and operational (mission-dependent) information and hence in addition to the number of roles in the team and the scope of the domain structure maintained, the message size depends on the specific mission type and its associated operational state aggregated through the management tree.

Also, mobility of UAVs has an effect on the communication overhead incurred by the management tree maintenance. Since discovery is performed on a single hop basis, in its initial form any edge of the management tree corresponds to only one physical link. However, as UAVs move around due to different reasons such as mission requirements, the tree depends on multi-hop communication thereby its edges being stretched over multiple physical links. This has a direct bearing over the network with respect to communication overhead as the state updates exchanged between a manager and (directly) managed role traverse multiple hops. In the worst case scenario, a management tree edge can be stretched as much as $n-1$ hops

thereby increasing the communication overhead.

8.2 Performance Evaluation of the Implementation

The prototype implementation was evaluated through experimentation on a testbed of Linux machines running Java to study the scalability of our framework with respect to the number of roles involved in a mission, the effect of cluster type failures, the effect of the depth of the management tree, the impact of large number of policies on the mission setup time, the effect of immediate role assignment and the time complexity of optimised role assignment. At the beginning of the simulation, the number and types of roles are changed in the mission class specification and the requisite number of instances of the framework are started on various machines. The experimental setup consisted of machines[1] on a Local Area Network[2]. We simulated different subnets by using IP filter policies. Each manager role was assigned to a separate machine and a different subnet, while other roles were running in parallel (with a maximum of 20 roles per machine). However, each role runs on its own management framework and hence the number of roles is the same as the number of UAVs, i.e., one machine simulates up to 20 UAVs.

8.2.1 Mission Setup Time

In this experiment, we fixed the depth of the management tree to 5 levels and compared its performance, with respect to mission setup time, with a centralised approach by varying the number of roles in the mission. The mission setup time includes the time taken for discovering the UAVs, assigning the roles, downloading the policies from the repository and loading them, and starting the roles. Figure 8.5 shows the result for 19 experiments plotted with a 95% confidence interval. The result illustrates that as the number of roles increases the hierarchical management approach outperforms the centralised one. Although the centralised approach performs better for missions involving smaller number of roles with respect to mission setup time, its ability to alleviate the problem of a single point of failure and increase availability, to cope with communication or geographic constraints as well as capture

[1] Intel(R) Core(TM)2 Duo CPU 3.00GHz, 4GB RAM.
[2] 1Gb Ethernet.

the management semantics in real-life missions makes the hierarchical approach more attractive.

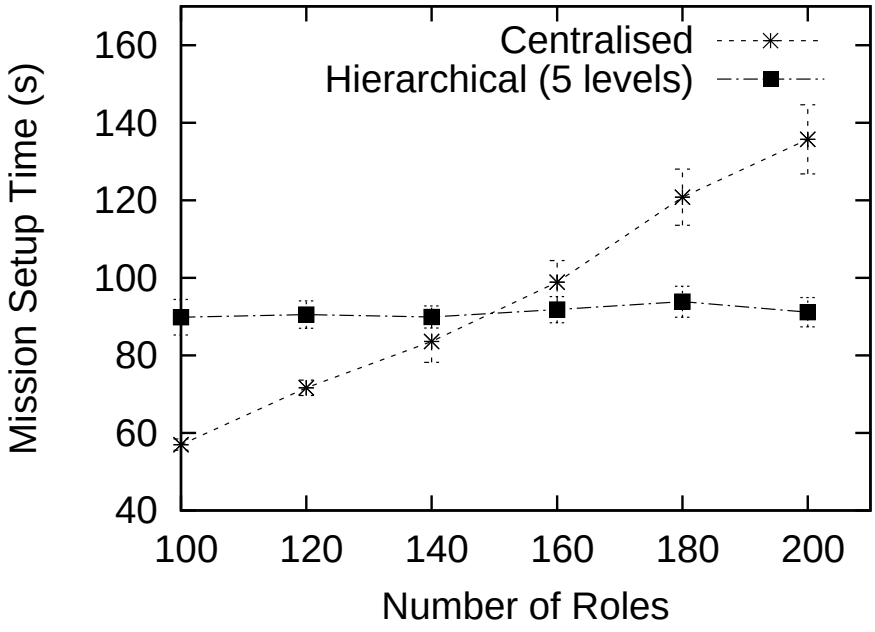


Figure 8.5: Comparison of Mission Setup Time between Centralised and Hierarchical Mission Management

8.2.2 Effect of the Depth of the Management Tree and Number of Roles

In this experiment, we varied the number of roles between 20 and 200 and the depth of the management tree between 1 (centralised) and 10. We then measured the time taken to setup the mission provided that UAVs consisting of all required capabilities are available during the mission startup time. Figure 8.6 shows the result for 25 experiments. For higher number of roles, the mission setup time decreases as we increase the depth of the tree as a result of load balancing. However, this trend stops and the setup times starts to increase slowly as the tree becomes very deep due to the delay in role assignment created by an increase in the number of hops. This behaviour suggests the existence of a ratio of number of roles to depth, for a given management tree, which guarantees a minimal mission setup time. For smaller number of roles, the mission setup time is minimal at depths 1 and 2 and after that the setup time increases with depth due to the overhead introduced by the added number of hops without any gain in load balancing as the number of roles is already

sufficiently small to be managed by few manager roles. It is also interesting to note that the depth at which the minimal setup time occurs increases as we increase the number of roles.

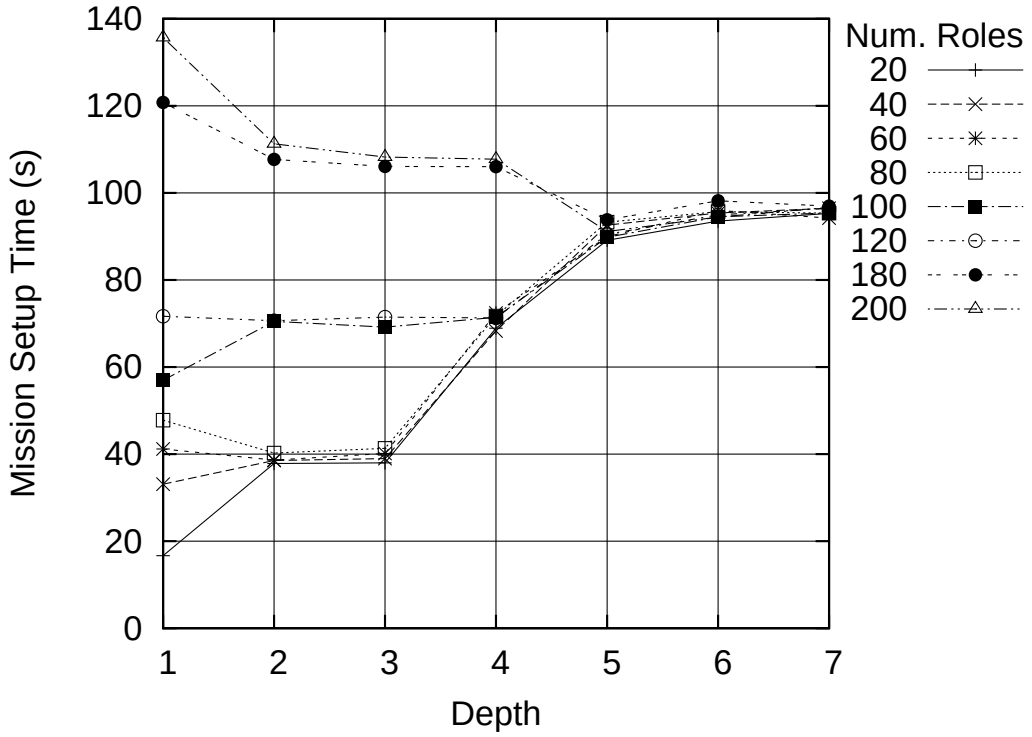


Figure 8.6: Measure of Time Complexity against the Depth of the Management Tree

8.2.3 Mean time to Reassign Roles after Failure

In this experiment, we study the response time of our mission management system when a cluster of failures occur as in typical disaster response or military scenarios it is likely that a group of UAVs could be affected by an event causing them all to fail. We used a mission specification that has 100 *Surveyor* roles and 100 *Aggregator* roles and a reassignment policy that dictates that whenever a *Surveyor* role fails an *Aggregator* role should be withdrawn from a working UAV and replaced by a *Surveyor* role. The results are shown in Figure 8.7. We note that the reassignment time scales linearly with the number of failed nodes.

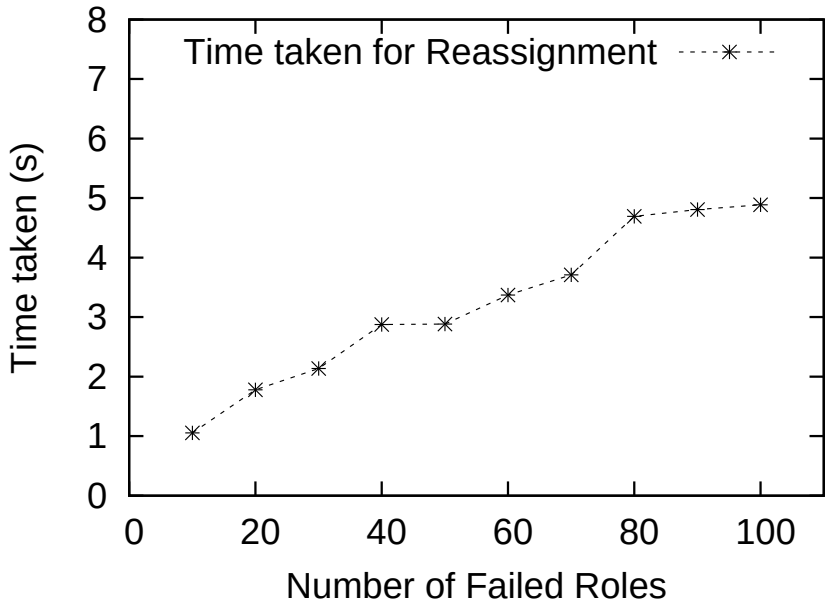


Figure 8.7: Measurement of Time Taken to Reassign Roles in a Cluster Failure Scenario

8.2.4 Mean time to Load Policies

Our framework depends heavily on policies, which are used for configuration, optimisation and adaptation. This gives rise to a large number of policies thus the need to evaluate the time complexity of loading these policies, which has a direct impact on the mission setup time.

In this experiment, we measured the time taken to fetch policies from the repository (web server) and load them. Figure 8.8 shows the result for 10 experiments plotted with a 95% confidence interval. The policy loading time increases linearly with the number of policies in the mission.

8.2.5 Comparison of the Immediate and Optimal Role Assignment Approaches

In this experiment, we compare the performance of the immediate and optimal role assignments with respect to time complexity and success rate of assignment. We considered a mission that has 10 role types and 10 UAV types as shown in Table 8.1. The capabilities of the first half of the UAV types, i.e., s_1 up to s_5, can each satisfy one role type (one out of r_1 up to r_5). Each of the second half of the UAV types (s_6 up

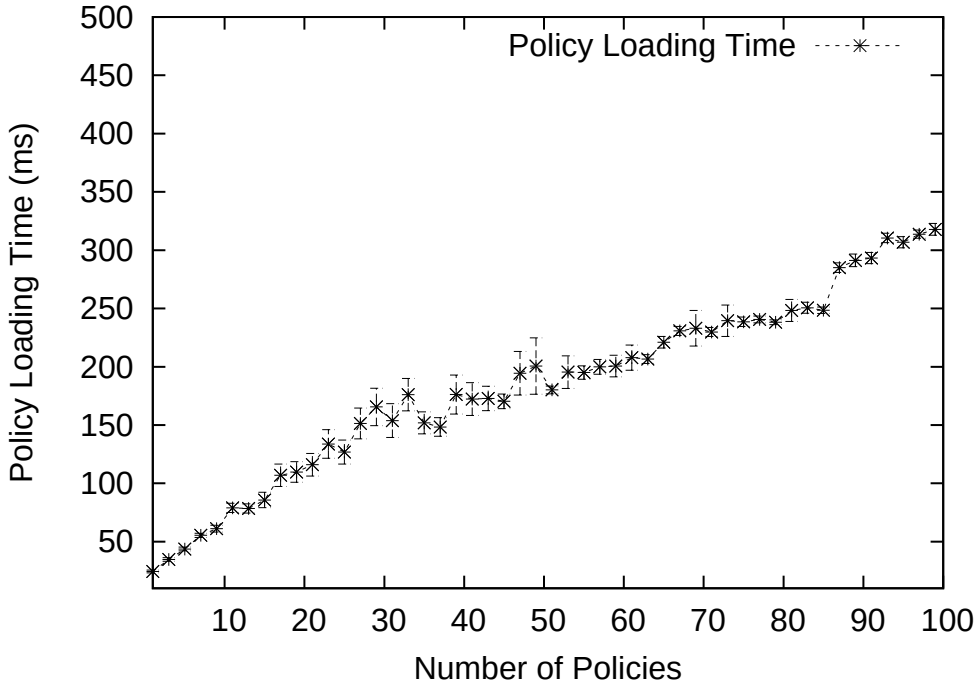


Figure 8.8: Policy Loading Time

to s_{10}) has capabilities that are diverse enough to satisfy the capability requirement of any of the first five role types (r_1 up to r_5) as well as one other role type out of the second half of the role types (r_6 up to r_{10}). Each of the first five role types can be assigned to one of the 6 possible UAV types while each of the rest of the role types can only be assigned to one UAV type.

If we perform an immediate role assignment on a first-discovered first-assigned basis, it may be the case that one or more of role types r_1 up to r_5 are assigned to a UAV of type s_6 up to s_{10} leading to an incomplete team since role types r_6 up to r_{10} can only be assigned to UAV types s_6 up to s_{10} respectively. The team can be complete only if the UAVs arrive in a sequence where all the first half of UAV types arrive before any of the UAV types of the second half, i.e., $s_1, s_2, s_3, s_4, s_5 \rightarrow s_6, s_7, s_8, s_9, s_{10}$. There are $5! * 5!$ possibilities of this arrival out of $10!$ possible arrival sequences. In the worst case scenario, all of the second half of the UAV types can arrive first leading to a team where only 50% of the roles are assigned.

In our experiment, we varied the number of roles and UAVs keeping their numbers equal and the role and UAV types and their proportions constant. In the case of the optimised role assignment, all the roles were assigned in all the experiments. Figure 8.9 shows the result of the immediate role assignment for 20 experiments plotted with

Role Type	**Required Capability**	**UAV Type**	**Provided Capability**
r_1	$\{c_1, c_2\}$	s_1	$\{c_1, c_2\}$
r_2	$\{c_1, c_3\}$	s_2	$\{c_1, c_3\}$
r_3	$\{c_1, c_4\}$	s_3	$\{c_1, c_4\}$
r_4	$\{c_1, c_5\}$	s_4	$\{c_1, c_5\}$
r_5	$\{c_1, c_6\}$	s_5	$\{c_1, c_6\}$
r_6	$\{c_1, c_7\}$	s_6	$\{c_1, c_2, c_3, c_4, c_5, c_6, c_7\}$
r_7	$\{c_8\}$	s_7	$\{c_1, c_2, c_3, c_4, c_5, c_6, c_8\}$
r_8	$\{c_9\}$	s_8	$\{c_1, c_2, c_3, c_4, c_5, c_6, c_9\}$
r_9	$\{c_{10}\}$	s_9	$\{c_1, c_2, c_3, c_4, c_5, c_6, c_{10}\}$
r_{10}	$\{c_{11}\}$	s_{10}	$\{c_1, c_2, c_3, c_4, c_5, c_6, c_{11}\}$

Table 8.1: Capability Requirements and Provisions

a 95% confidence interval where *Role Assignment Success Rate* is the percentage of roles assigned to UAVs out of the total number of available roles. The plot labelled as *Immediate* shows the observed success rate of the immediate role assignment scheme during the experiment. Note that the success rate of this assignment scheme varies as it depends on UAV arrival sequences. The plot labelled as *Optimised* shows the observed success rate of the optimised role assignment scheme. Theoretically, this success rate can also be achieved in the immediate role assignment scheme (best case scenario) provided that the UAVs arrive in an ideal sequence, i.e., all the first half of UAV types (shown in Table 8.1) arrive before any of the UAV types of the second half, which is impractical. The plot labelled as *Immediate (worst case)* shows what the success rate would be if the UAVs arrive in the worst possible sequence, i.e., all of the second half of the UAV types (shown in Table 8.1) arrive before the first half.

Figure 8.10 shows the time taken for assignment for both immediate and optimised role assignments. The assignment time is measured from the time the first UAV is discovered to either the time the last role is assigned (in the case of the optimised assignment) or all UAVs have been discovered and all roles that can be assigned are assigned (in the case of the immediate assignment). The polynomial time complexity of the optimised assignment is attributed to the $O(n^3)$ algorithm[3] we used to compute the minimum-cost maximum bipartite matching of the assignment graph. Although the assignment algorithm, for the sake of the experiment, starts optimisation after

[3]n is the number of roles/UAVs.

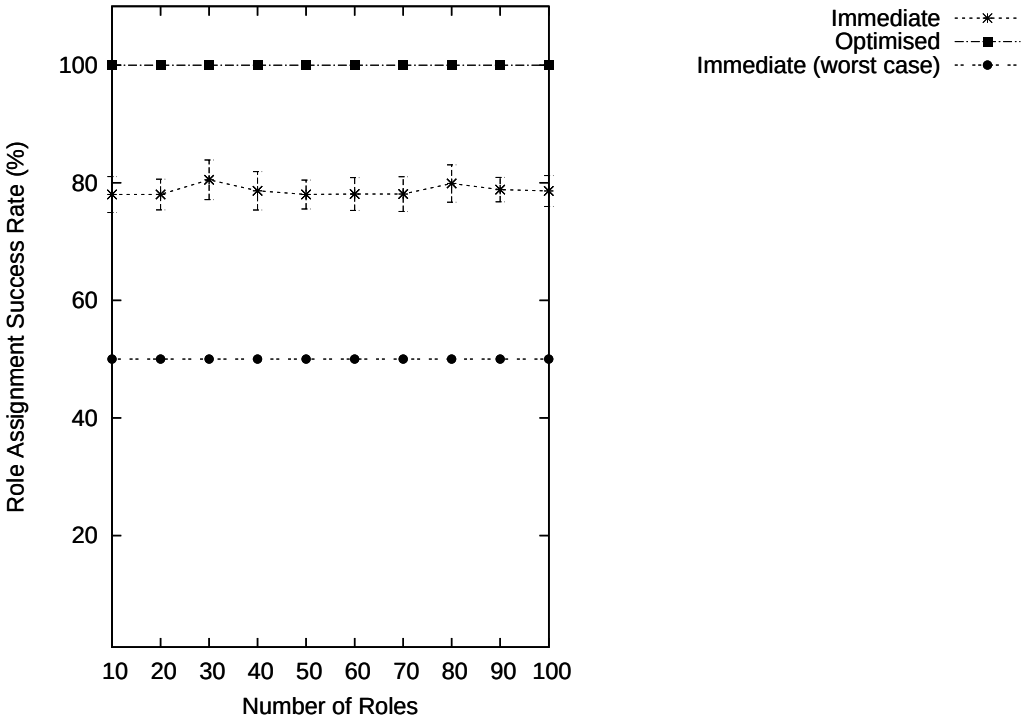


Figure 8.9: Role Assignment Success Rate

all UAVs are discovered, in reality the waiting period is set by a policy and there is a tradeoff between the length of this period and the success rate of the assignment.

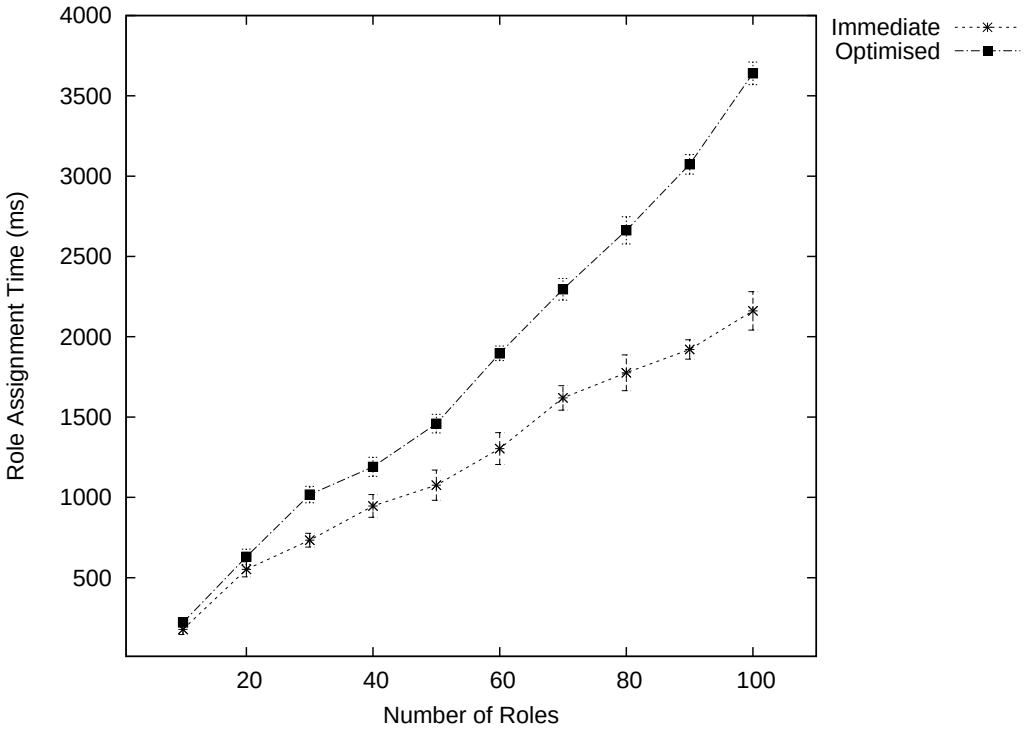


Figure 8.10: Time Complexity of the Optimised and Immediate Assignment Algorithms

8.2.6 Evaluation of Communication Management

The communication management scheme was implemented using the Webots mobile robotics simulator [Ltd], which is a prototyping environment for modelling, programming and simulating mobile robots. A total of five UAVs, with a management tree as shown in Figure 4.6, were used for this experiment

Evaluation of Approach 1: Adapt Movement to Maintain Communication

In the first experiment (Figure 8.11), the effect of range threshold was evaluated with respect to the speed of the UAVs, while the second experiment (Figure 8.12) evaluated the update time versus the speed of the UAVs. The success rate is defined as the number of UAVs that successfully manage to follow the lead UAV to its destination (including the lead UAV itself). For the purpose of the experiments, the *Surveyor* was acting as the leader UAV. For the value of speed, a magnitude of 1 denotes a speed of $4.5\ mm/s$. For the first experiment, the update time is set to $2s$, while for the second experiment, the range threshold is set to 75% (i.e., 75% of the communication range).

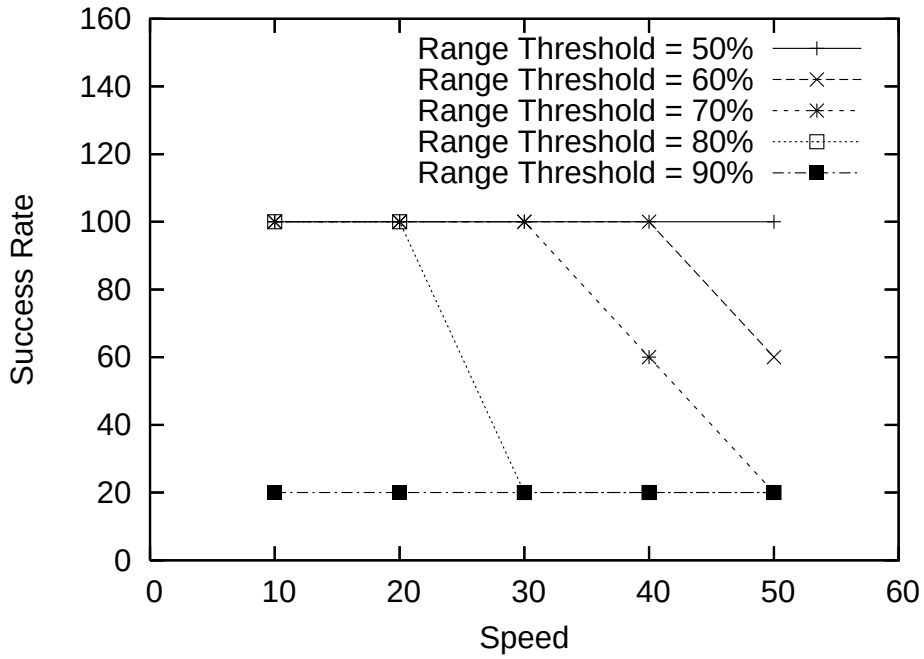


Figure 8.11: Communication Management (Changing the Range Threshold)

From Figure 8.11, we see that the range threshold (T_R) has a significant impact on the performance. As the value of T_R increases, fewer and fewer UAVs are able to

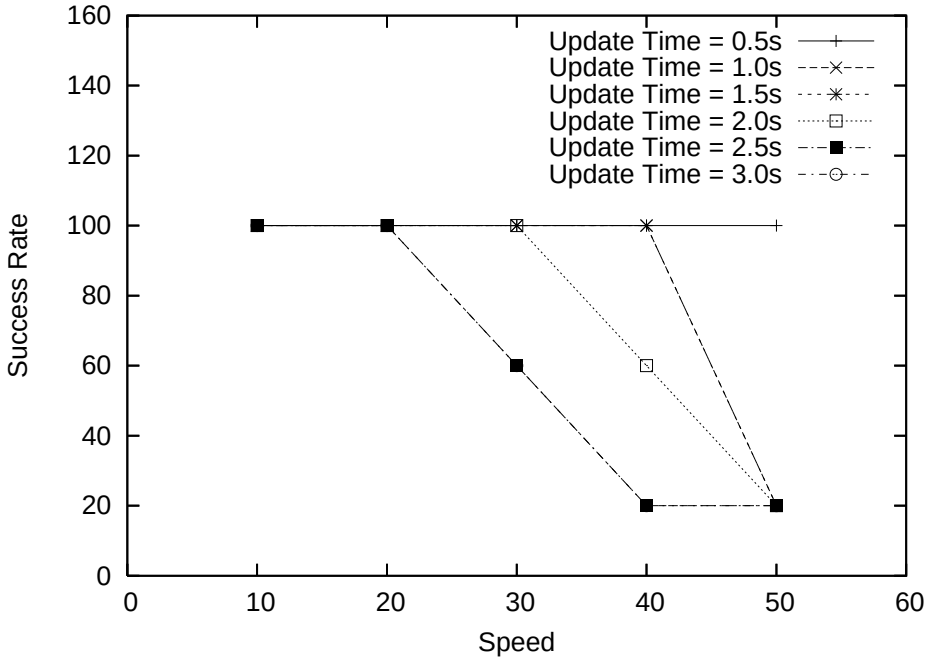


Figure 8.12: Communication Management (Changing the Update Time)

follow the leader. This is especially true in the case when the speed is greater than 30, since only the leader is able to reach its destination. Setting the value of T_R much lower (50%) enables all UAVs to reach the destination, but this (1) gives less time for follower UAVs to complete their task at their current position, and (2) may be detrimental since it results in the leader being followed too closely and may result in a cluster failure (e.g., due to a hazardous terrain). Hence, instead of setting the value of T_R a priori, it is better to set the value dynamically based on the current speed and task status of the target UAV. We did not consider a model for computing T_R in this adaptive manner. However, we have made T_R adaptable during mission execution thereby enabling extension of the framework.

From Figure 8.12, we can see that the change in update time adversely affects the UAVs when they are travelling at a high speed. This is to be expected since the follower UAV uses the location updates of its leader to map its path. Having a small update rate ($0.5s$) results in all UAVs following the leader to the destination. However, this has an adverse effect on network traffic and the battery life of the UAVs due to the excess communication.

Evaluation of Approach 2: Rendezvous to Restore Communication

In this experiment, we evaluate the speed of the UAVs and the time taken by them to reach the rendezvous area with respect to the rendezvous time. We evaluate three schemes: (i) the UAVs continue to the rendezvous area at their current speed, (ii) the UAVs continue at an average speed (an estimate of the speed required to reach the rendezvous point) , and (iii) the UAVs continue at their maximum allowed speed. For the value of speed, a magnitude of 1 denotes a speed of $4.5\ mm/s$. The speed of the UAV (the *Surveyor* for the purpose of this experiment) that departs and causes the rendezvous setup was set to 30. The range threshold is set to 75%. The rendezvous time is varied between 10 and 50 seconds and the maximum speed is set to 50.

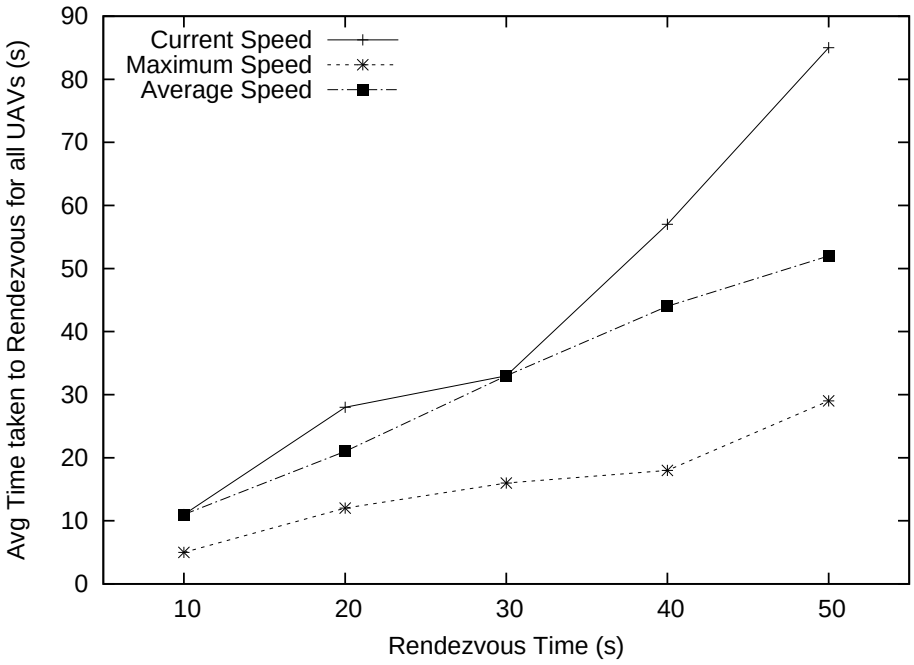


Figure 8.13: Communication Management (Rendezvous Time)

From Figure 8.2.6, we can see that as the rendezvous time increases, the time taken in all the schemes also increases. This is to be expected, since the rendezvous area is a little bit further away each time. The result also shows that the UAVs reach the rendezvous area on time provided that they travel at the average speed. This would allow faster UAVs to conserve their energy by reducing their speed. Slower UAVs would have to increase their speed, but since they are travelling at the average speed, the energy consumed is not very much. Also, we can see from the result that there is a significant difference in the time taken when the UAVs move at their

top speed. This is especially useful in the scenario when it is critical to reach the rendezvous area as soon as possible.

8.3 Critical Evaluation of the Framework

Our framework enables the specification of adaptable and reusable missions for autonomous systems. In accordance with the mission specification, a team is formed dynamically by discovering UAVs and assigning roles to them in an optimal manner. In addition, the framework deals with communication link maintenance among team members and recovery from failure. The management task is distributed in that all manager roles in the team perform role assignment and decisions pertaining to adaptation in relation to roles managed by them. This distribution enables the framework to meet application specific requirements such as the hierarchical organisation of a search and rescue team. As illustrated in the performance evaluation, the framework is scalable with respect to the number of roles and policies involved in the mission, and optimal with respect to role assignments. From the analysis of the message complexity of the management tree formation and maintenance, we observe that the communication overhead is significant. Although it is shown that the overhead can be decreased for missions that do not involve interactions between all of the mission roles, enabling a more fine grained control of this overhead through policies is required to make the framework feasible in missions in which communication links have low bandwidth. The primary limitation of our framework is the lack of automated or semi-automated mission refinement. The refinement of a mission statement into roles and policies is an intensive manual or semi-manual task that needs much research to create tools that may assist in this process. Work is being done within our group in this direction. The refinement of general high-level policies into more specific policies is considered in [BLMR04, BLR+06] that provides tool support and partial policy refinement automation. Our framework also does not consider policy conflicts. A mechanism for analysing policies in order to detect conflicts [MS94, LS99, CLM+09] is necessary.

The framework assigns UAVs to roles based on their capabilities and credentials. However, a selfish autonomous system might advertise less capabilities in order to avoid duty. Similarly, a malicious autonomous system might advertise more capa-

bility and be assigned to a role it cannot perform in order to sabotage the mission. Although UAVs are authenticated before they are assigned to any role, selfish or malicious systems which have the necessary credentials but lack the goodwill to achieve the goals of the mission can exploit the framework after the authentication phase. The implicit assumption that all authenticated UAVs are trusted is detrimental in some application scenarios and hence the framework could be enhanced with a trust management system. Research in open multi-agent systems [RHJ05, HJS06] has looked into this issue.

Although the framework enables the specification of role behaviours using policies and the formation of a team in accordance with a high-level mission specification, the lack of a monitoring entity limits the spectrum of adaptation that would have been possible. Both the role and team behaviour are readily adaptable through policies. However, there is a need for an entity that monitors the performance of the team and generates events to initiate the adaptation, and policies to make a higher level of adaptation where the policy defining the behaviour of the role or the team itself is adapted.

The framework does not require pre-mission-setup knowledge of the capabilities requirement of the roles or the capabilities provision of the UAVs. However, it is implicitly assumed that there is an agreement on the vocabulary of capabilities, more generally on the ontology of capabilities and services. For example, when a role puts forward *gps* as a required capability, it is assumed that (1) all UAVs advertising *gps* in their capability mean the same thing and (2) any advertisement that does not contain the keyword *gps* does not satisfy the requirement of the role. An explicit definition of ontology of UAVs would make the team formation seamless and also facilitate semantic based capability matching, i.e., allowing for the more realistic setting that capabilities with differing names may satisfy the same requirement.

8.4 Summary

In this chapter, we have presented analysis of the communication overhead incurred by the management tree, performance evaluation of the prototype implementation and a critical evaluation of the framework.

We have discussed the limitations and vulnerabilities of the framework. The framework is vulnerable to rogue capability advertisements by selfish or malicious UAVs. It also lacks an automated or semi-automated means of mission refinement. The XML based capability description and matching assumes an implicit ontology and does not support semantic based matching.

We have shown through the evaluation of the proof-of-concept implementation that the framework is scalable with respect to the number of roles and policies involved in a mission. We have also shown that the framework performs optimal role assignments and can cope with simultaneous failures of a large number of UAVs.

Chapter 9

Conclusions

In this chapter, we will summarise the thesis achievements and compare them against the requirements for a self-management framework stated in the introduction of this thesis. We also indicate directions for future work based on our critical analysis of the framework and conclude the thesis.

9.1 Achievements

The first requirement for the self-management framework is support for a mission specification approach that enables specifying missions in an adaptable and reusable manner for teams of mobile autonomous systems. We have proposed a novel role-based mission specification for teams of mobile autonomous systems where the behaviour of individual members as well as the team as a whole is specified using policies thereby allowing us to enforce team (organisational) as well as individual norms. The team can be adapted in two ways: a policy can change the behaviour of the team in response to an event or a policy governing the behaviour of the team can itself be adapted.

A mission for a team of mobile autonomous systems is specified in three levels, namely policy, mission class and mission-class instance specifications allowing for reuse of specifications. This enables a policy to be used with different mission classes and a mission class to be used with different mission-class instance specifications

tailored to specific scenarios.

The second requirement for the framework is support for secure and optimal formation, usage and maintenance of dynamic teams. In our framework, autonomous systems are discovered, authenticated and assigned to roles dynamically and when they depart or fail they are replaced with other autonomous systems. The role assignment is based on the credentials and capabilities of the autonomous system, and the requirement of the role. Instead of immediately assigning roles in a greedy manner, we delay the assignments and attempt to perform utility-function based optimisation on the set of discovered systems and required roles. The framework detects intermittent communication link disconnection and permanent link or system failures and reconfigures itself in accordance with failure management policies. The behaviour of the team management part of the framework can also be adapted using policies. For example, one can change discovery, role assignment, optimisation and failure parameters at run time.

The final requirement is support for communication link maintenance among mobile autonomous systems. The communication layer of the framework actively tries to maintain communication between team members using two techniques: adapting the movement of autonomous systems so that each node always has one or more other nodes within communication range to maintain an ad hoc network, and setting up a rendezvous within a defined rendezvous area at a specified time in order to enable communication at regular intervals. In the event that a UAV is unable to reach the rendezvous area, it is assumed to have failed and the failure management scheme in the team layer is used to recover.

The applicability of the framework to real-life problems was illustrated using a search and rescue scenario. We have described the mission specification process, which involves mission, role and policy refinement.

We have implemented our framework using the Ponder2 policy toolkit and tested the prototype implementation using a search and rescue mission with the Koala robots, in a controlled environment. We have also evaluated the performance of the framework and shown that it is efficient with respect to mission setup time, scalable with respect to the number of roles and policies involved in the mission, and optimal with respect to role assignments.

9.2 Future Work

Although the mission specification in our framework is adaptable and reusable, when done from scratch the specification process involves manual refinement of mission statements into policies and roles, which are used as the building blocks of the specification. Developing tools which could assist in this process would make the framework more applicable in real-life scenarios.

This thesis has only addressed reconfiguration triggered by the need to adapt to failure; further work is needed to investigate the impact of reconfiguration that is driven by the necessity to improve the performance of the team. A monitoring service possibly with a learning module would help in generating the necessary events for adaptation and selecting appropriate mission parameters based on experience. Utility functions, which can be used to measure the quality of possible reconfigurations in order to compare and select one out of the many possibilities, are necessary.

In order to be able to match capabilities that have syntactic differences, a semantic-based capability matching approach, with an ontology for mobile autonomous system capabilities needs to be developed. The capability description scheme needs to be enhanced to support semantic-based matching.

In a dynamic and open team such as the one considered in this thesis, it is impractical to assume that all authenticated autonomous systems are trusted. Malicious systems can sabotage the mission in a number of ways, consequently the framework could be enhanced with a trust management system.

An autonomous system may be captured or, as a result of a fault, start behaving anomalously. It is necessary to be able to detect anomalous behaviour, take action to isolate the anomalous system from the team and possibly distribute new secret keys to other team members.

9.3 Closing Remarks

The framework proposed in this thesis has been motivated by the need for enabling self-management in the increasingly ubiquitous mobile autonomous systems. The

novel approaches we proposed for mission specification, team formation and communication link maintenance will facilitate the application of mobile autonomous systems such as UAVs in real-life missions.

Appendices

Appendix A

PonderTalk Basic Types and Operations

As discussed in Chapter 2, PonderTalk has basic (built in) objects that are *Array*, *Number*, *String*, *Boolean*, *Nil*, *Hash*, *Xml* and the special object *Block*. Since the *Array* and *Hash* objects support a number of operations and these operations are used in example policies in the thesis, a listing and description of the operations is shown here.

Operation	Remark
do: aBlock	Takes a block and executes the block once for each entry in the array and returns the result of the last block executed. The arguments to the block are the name of the entry and the value of the entry.
collect: aBlock	Takes a block and executes the block once for each entry in the array. The arguments to the block are the name of the entry and the value of the entry. The result of each block is collected and returned in an array.
add: aP2Object	Adds aP2Object to the array and returns itself.
addAll: anArray	Adds all object in the array anArray to the array and returns itself
at: anIndex	Returns the object at anIndex.
at: anIndex **put:** aP2Object	Inserts aP2Object in the array at anIndex and returns the insterted object – aP2Object.
remove: anIndex	Returns the value at anIndex and removes it from the array.
removeObject: anObject	Removes anObject from the array. All copies of anObject will be removed. Returns true if one or more were removed.
removeAll	Removes all objects stored in the array and returns itself.
hasObject: anObject	Returns true if anObject is in the array.
size	Returns the number of elements in the array.

Table A.1: Messages (operations) Supported by PonderTalk's Array Object

Operation	Remark
do: aBlock	Takes a block and executes the block once for each entry in the table and returns the result of the last block executed. The arguments to the block are the name of the entry and the value of the entry.
collect: aBlock	Takes a block and executes the block once for each entry in the table. The arguments to the block are the name of the entry and the value of the entry. The result of each block is collected and returned in an array.
listNames	Returns an array containing the names of all the entries of the table.
listObjects	Returns an array of objects containing all the entries in the table.
asArray	Returns an array containing a flattened hash table in the form of name, value, name, value ... entries.
at: aKey **put:** anObject	Store anObject in the table with aKey and returns the stored object – anObject. If aKey already exists, the previous value is overridden.
remove: aKey	Returns the value associated with the given key and removes it from the table. Returns Null if it is not found.
removeObject: anObject	Removes anObject from the table. All copies of anObject will be removed. Returns true if one or more were removed.
removeAll	Removes all objects stored in the table and returns itself.
has: aKey	Returns true if the given key exists in the table otherwise false.
hasObject: anObject	Returns true if anObject is in the table.
at: aKey **ifAbsent**: aBlock	Returns the value associated with the given key. If not found, evaluate block (with no arguments) and return its result.
at: aKey	Returns the value associated with the given key. Throws a Ponder2ArgumentException error if not found.
size	Returns the number of elements in the table.

Table A.2: Messages (operations) Supported by PonderTalk's Hash-table Object

Appendix B

Role Specifications and Policies

B.1 Role Specifications

Role specifications of the case study search and rescue mission are shown in this section.

```
<xml>
  <role name='Aggregator'>
    <policy uri='http://192.168.0.1/search_rescue/policy/aggregator'/>
    <tasks>
          <task name='AssessRisk'>
        <expose>
          <operations>
            <operation name='assessRisk'>
              <argument>
                <name>survivorLocation</name>
                <type>dmrc.util.Location</type>
              </argument>
            </operation>
          </operations>
          <notifications>
            <notification name='riskAssessed'>
              <attribute name='name'/>
              <attribute name='survivorLocation'/>
              <attribute name='riskLevel'/>
            </notification>
          </notifications>
        </expose>
      </task>
```

Figure B.1: Search & Rescue Role Specification - Aggregator Role (Part 1)

```
24        <task name='BuildMap'>
25          <expose>
26            <operations>
27              <operation name='setDisasterArea'>
28                <argument>
29                  <name>area</name>
30                  <type>dmrc.util.Area</type>
31                  <name>base</name>
32                  <type>dmrc.util.Location</type>
33                </argument>
34              </operation>
35              <operation name='updateMap'>
36                <argument>
37                  <name>location</name>
38                  <type>dmrc.util.Location</type>
39                </argument>
40                <argument>
41                  <name>hazard</name>
42                  <type>dmrc.util.HazardType</type>
43                </argument>
44              </operation>
45              <operation name='getMap'>
46                <argument/>
47                <result>
48                  <name>result</name>
49                  <type>dmrc.util.DmrcMap</type>
50                </result>
51              </operation>
52              <operation name='uploadVisualDcoument'>
53                <argument>
54                  <name>visualDocument</name>
55                  <type>dmrc.util.Media</type>
56                </argument>
57              </operation>
58            </operations>
59          </expose>
60        </task>
61      </tasks>
62      <expose>
63        <notifications>
64          <notification name='rescueStatus'>
65            <attribute name='name'/>
66            <attribute name='coverage'/>
67            <attribute name='rescued'/>
68          </notification>
69        </notifications>
70      </expose>
71      <require>
72        <operations>
73          <operation name='setArea'>
74            <argument>
75              <name>area</name>
76              <type>dmrc.util.Area</type>
77            </argument>
78          </operation>
79        </operations>
80        <notifications>
81          <notification name='survivorRescued'>
82            <attribute name='name'/>
83            <attribute name='survivorLocation'/>
84          </notification>
85          <notification name='batteryLevel'>
86            <attribute name='name'/>
87            <attribute name='level'/>
88          </notification>
89        </notifications>
90      </require>
91      <capability>
92        <require>
93          <type>motion</type>
94        </require>
95      </capability>
96    </role>
97  </xml>
```

Figure B.2: Search & Rescue Role Specification - Aggregator Role (Part 2)

```
<xml>
  <role name='Hdetector'>
    <policy uri='http://192.168.0.1/search_rescue/policy/hdetector'/>
    <tasks>
      <task name='DetectHazard'>
        <expose>
          <operations>
            <operation name='setArea'>
              <argument>
                <name>area</name>
                <type>dmrc.util.Area</type>
                <name>base</name>
                <type>dmrc.util.Location</type>
              </argument>
            </operation>
          </operations>
          <notifications>
            <notification name='hazardDetected'>
              <attribute name='name'/>
              <attribute name='hazardLocation'/>
              <attribute name='hazardType'/>
            </notification>
          </notifications>
        </expose>
      </task>
    </tasks>
    <expose/>
    <require>
      <operations>
        <operation name='updateMap'>
          <argument>
            <name>location</name>
            <type>dmrc.util.Location</type>
          </argument>
          <argument>
            <name>hazard</name>
            <type>dmrc.util.HazardType</type>
          </argument>
        </operation>
      </operations>
    </require>
    <capability>
      <require>
        <type>motion</type>
        <type>chemicalDetection</type>
        <type>biologicalDetection</type>
      </require>
    </capability>
  </role>
</xml>
```

Figure B.3: Search & Rescue Role Specification - Hazard-detector Role

```
<xml>
  <role name='Medic'>
    <policy uri='http://192.168.0.1/search_rescue/policy/medic'/>
    <tasks>
      <task name='AssistSurvivor'>
        <expose>
          <operations>
            <operation name='assist'>
              <argument>
                <name>survivorLocation</name>
                <type>dmrc.util.Location</type>
              </argument>
            </operation>
            <operation name='measureMetric'>
              <argument>
                <name>roleId</name>
                <type>dmrc.util.RoleIdentity</type>
              </argument>
              <argument>
                <name>metricType</name>
                <type>String</type>
              </argument>
              <result>
                <name>metricValue</name>
                <type>Double</type>
              </result>
            </operation>
          </operations>
          <notifications>
            <notification name='survivorAssissted'>
              <attribute name='name'/>
              <attribute name='survivorLocation'/>
            </notification>
          </notifications>
        </expose>
      </task>
    </tasks>
    <expose/>
    <require/>
    <capability>
      <require>
        <type>motion</type>
        <type>medical</type>
      </require>
    </capability>
  </role>
</xml>
```

Figure B.4: Search & Rescue Role Specification - Medic Role

```
<xml>
  <role name='Transporter'>
    <policy uri='http://192.168.0.1/search_rescue/policy/transporter'/>
    <tasks>
      <task name='Transport'>
        <expose>
          <operations>
            <operation name='transport'>
              <argument>
                <name>survivorLocation</name>
                <type>dmrc.util.Location</type>
              </argument>
              <argument>
                <name>destination</name>
                <type>dmrc.util.Location</type>
              </argument>
            </operation>
            <operation name='measureMetric'>
              <argument>
                <name>roleId</name>
                <type>dmrc.util.RoleIdentity</type>
              </argument>
              <argument>
                <name>metricType</name>
                <type>String</type>
              </argument>
              <result>
                <name>metricValue</name>
                <type>Double</type>
              </result>
            </operation>
          </operations>
          <notifications>
            <notification name='survivorTransported'>
              <attribute name='name'/>
              <attribute name='survivorOriginalLocation'/>
              <attribute name='survivorNewLocation'/>
            </notification>
          </notifications>
        </expose>
      </task>
    </tasks>
    <expose/>
    <require/>
    <capability>
      <require>
        <type>motion</type>
        <type>lifting</type>
      </require>
    </capability>
  </role>
</xml>
```

Figure B.5: Search & Rescue Role Specification - Transporter Role

```
<xml>
  <role name='Relay'>
    <policy uri='http://192.168.0.1/search_rescue/policy/relay'/>
    <tasks>
      <task name='RelayFunction'>
        <expose>
          <operations>
            <operation name='setMembers'>
              <argument>
                <name>uavList</name>
                <type>dmrc.util.List</type>
              </argument>
            </operation>
          </operations>
          <notifications>
            <notification name='memberOutofRange'>
              <attribute name='name'/>
              <attribute name='uav'/>
            </notification>
          </notifications>
        </expose>
      </task>
    </tasks>
    <expose/>
    <require/>
    <capability>
      <require>
        <type>motion</type>
        <type>longrangecom</type>
      </require>
    </capability>
  </role>
</xml>
```

Figure B.6: Search & Rescue Role Specification - Relay Role

```
<xml>
  <role name='Commander'>
    <policy uri='http://192.168.0.1/search_rescue/policy/commander'/>
    <tasks>
      <task name='ManageEarthQuakeDisaster'>
        <expose>
          <operations>
            <operation name='setDisasterArea'>
              <argument>
                <name>area</name>
                <type>dmrc.util.Area</type>
                <name>base</name>
                <type>dmrc.util.Location</type>
              </argument>
            </operation>
          </operations>
          <notifications>
            <notification name='memberOutofRange'>
              <attribute name='name'/>
              <attribute name='uav'/>
            </notification>
          </notifications>
        </expose>
      </task>
    </tasks>
    <expose/>
    <require>
      <require>
        <operations>
          <operation name='setMembers'>
            <argument>
              <name>uavList</name>
              <type>dmrc.util.List</type>
            </argument>
          </operation>
        </operations>
        <notifications>
          <notification name='rescueStatus'>
            <attribute name='name'/>
            <attribute name='coverage'/>
            <attribute name='rescued'/>
          </notification>
        </notifications>
      </require>
    </require>
    <capability>
      <require>
        <type>motion</type>
        <type>longrangecom</type>
      </require>
    </capability>
  </role>
</xml>
```

Figure B.7: Search & Rescue Role Specification - Commander Role

B.2 Bootstrapping the Management Framework

The PonderTalk statements for bootstrapping the management framework are shown here.

```
//Import the Domain code and create the default domains.
domainFactory := root load: "Domain".
root
       at: "factory" put: domainFactory create;
       at: "policy" put: domainFactory create;
       at: "event" put: domainFactory create;
       at: "task" put: domainFactory create;
       at: "role" put: domainFactory create;
       at: "utility" put: domainFactory create.
//Put the domain factory into the factory directory.
root/factory at: "domain" put: domainFactory.
//Import event and policy factories.
root/factory.
        at: "event" put: ( root load: "EventTemplate" );
        at: "ecapolicy" put: ( root load: "ObligationPolicy" );
        at: "authpolicy" put: ( root load: "AuthorisationPolicy" ).
//Import and create the Event bus.
root/factory at:"eventservice" put: (root load: "SMCCore.EventService").
root at:"eventbus" put: (root/factory/eventservice create).
//Import and create the Mission entity.
root/factory at: "mission" put:(root load:"dmrc.mission.MissionClass").
root at: "mission" put:(root/factory/mission create).
//Import mission-specific role code. Creating
//the role objects depends on the role the
//UAV would enact and is done by a policy.
//UAVs may not have loaded all types of roles
//of a mission, upon assignment they can load the necessary role code.
//Surveyor role code
root/factory at:"surveyor" put:(root load: "dmrc.role.Surveyor").
root/factory at:"surveyorexternal" put:(root load: "dmrc.role.SurveyorExternal").
//Aggregator role code
root/factory at:"aggregator" put:(root load: "dmrc.role.Aggregator").
root/factory at:"aggregatorexternal"
    put:(root load: "dmrc.role.AggregatorExternal").
//Import and create the Role Manager.
root/factory at: "rolemanager" put:(root load:"dmrc.mission.RoleManager").
root at: "rolemanager" put:(root/factory/rolemanager create).
//Import and create the Capability Manager elements.
root/factory at:"capadvertiser" put: (root load:"dmrc.team.CapabilityAdvertiser").
root at:"capadvertiser" put:(root/factory/capadvertiser create).
root/factory at:"capability" put:(root load: "dmrc.team.Capability").
root at:"capability" put:(root/factory/capability create).
//Import the Discovery and Optimiser code.
//Creating these services is done by a
//policy depending on the role of the UAV.
root/factory at:"discovery" put: (root load: "dmrc.team.UxvDiscovery").
root/factory at:"optimiser" put: (root load: "dmrc.optim.Optimiser").
root at:"optimiser" put: (root/factory/optimiser create).
//Import and create the Message router.
root/factory at:"messagerouter" put: (root load: "dmrc.com.MessageRouter").
root at:"messagerouter" put: (root/factory/messagerouter create).
//Import and create the reliable Message sender.
//root/factory at:"messagesender" put: (root load: "dmrc.com.MessageSender").
//root at:"messagesender" put: (root/factory/messagesender create).
//Import the Communication service code.
root/factory at:"communication" put: (root load: "dmrc.com.Communication").
//Creating the service involves using credentials of the UAV (certificate & key)
root at:"communication" put: (root/factory/communication
    createSecure: "uav.der" key:"uavkey.der").
//Import and create the communication link maintainer.
root/factory at:"comlinkmaintainer" put:(root load:"dmrc.com.ComLinkMaintenance").
root at:"comlinkmaintainer" put:(root/factory/comlinkmaintainer create).
//Import and create the PonderTalk interpreter.
root/factory at:"pondertalk" put:(root load: "PonderTalk").
root at: "pondertalk" put:(root/factory/pondertalk create).
```

Figure B.8: Bootstrapping the Management Framework

B.3 Policies

Policies of the search and rescue mission (case study) that were not included in Chapter 6 are shown in this section.

```
policy := root/factory/ecapolicy create.
policy event: /event/newRelay;
action: [ :name :role :instance|
//Create tasks.
((root/task asHash) has: "rangeextend") ifFalse: [
root/task at: "rangeextend" put:((root load: "dmrc.task.RangeExtend") create)].
//Configure tasks.
root/task/rangeextend bindMotionTask:(/root/task/motion).
//Bind tasks.
(root/role resolve: (role+"/"+instance))
bindRangeExtendTask: (root/task/rangeextend).
root/comlinkmaintainer setLeader: "surveyor".
].
policy active: true.
```

Figure B.9: Search & Rescue Mission – Relay Role Policies

```
policy := root/factory/ecapolicy create.
policy event: /event/newHdetector;
action: [ :name :role :instance|
//Create tasks.
((root/task asHash) has: "explore") ifFalse: [
root/task at: "explore" put:((root load: "dmrc.task.Explore") create)].
((root/task asHash) has: "detecthazard") ifFalse: [
root/task at: "detecthazard" put:((root load: "dmrc.task.DetectHazard") create)].
//Configure tasks.
root/task/detecthazard bindExploreTask: (/root/task/explore).
root/task/explore bindCameraTask:(/root/task/camera).
root/task/explore bindMotionTask:(/root/task/motion).
root/task/explore bindBuildMapTask: (/root/task/buildmap).
//Bind tasks.
(root/role resolve: (role+"/"+instance))
                    bindDetectHazardTask: (root/task/detecthazard).
root/comlinkmaintainer setLeader: "surveyor".].
policy active: true.
```

Figure B.10: Search & Rescue Mission – Hazard-detector Role Policies

```
policy := root/factory/ecapolicy create.
policy event: /event/newMedic;
action: [ :name :role :instance|
//Create tasks.
((root/task asHash) has: "assistsurvivor") ifFalse:[root/task at: "assistsurvivor"
     put:((root load: "dmrc.task.AssistSurvivor") create)].
//Configure tasks.
root/task/assistsurvivor bindMotionTask:(/root/task/motion).
//Bind tasks.
(root/role resolve: (role+"/"+instance))
                    bindAssistSurvivorTask: (root/task/assistsurvivor).
root/comlinkmaintainer setLeader: "surveyor".].
policy active: true.
```

Figure B.11: Search & Rescue Mission – Medic Role Policies

```
13 policy := root/factory/ecapolicy create.
14 policy event: /event/newTransporter;
15 action: [ :name :role :instance|
16 //Create tasks.
17 ((root/task asHash) has: "transport") ifFalse: [
18 root/task at: "transport" put:((root load: "dmrc.task.Transport") create)].
19 //Configure tasks.
20 root/task/transport bindMotionTask:(/root/task/motion).
21 //Bind tasks.
22 (root/role resolve: (role+"/"+instance))
23                     bindTransportTask: (root/task/transport).
24 root/comlinkmaintainer setLeader: "surveyor".].
25 policy active: true.
```

Figure B.12: Search & Rescue Mission – Transporter Role Policies

B.4 A Note about Managed Objects

The means for controlling the entities of the framework, which are implemented as Ponder2 Managed objects, using policies are PonderTalk messages (discussed in Chapter 2, Section 2.6.1). For example, in some of the sample policies (e.g., Figure 6.15, line 27) shown in the thesis, we have seen the statement *root/discovery adduav: uav cap: lowLevelCap* in the action part of the policies. The policies were able to perform this operation – *"adduav uav: cap: lowLevelCap"* – that is used to add newly discovered UAVs to the assignment graph, because the discovery service that is implemented by the $UxvDiscovery$ class is a managed object that, among other messages, can be sent the $"adduav : cap : "$ PonderTalk message. Classes, such as $UxvDiscovery$, implementing managed objects, such as the discovery service, which need to receive PonderTalk messages annotate the corresponding method with Ponder2's **@Ponder2op** annotation in order to tie the PonderTalk message to the corresponding Java method as shown below.

```
@Ponder2op("adduav:cap:")
public void addUav(String uav, String capability){
addUxv(uav, capability);
}
```

Compilation of the managed object results in an adaptor object that maps the PonderTalk messages to the Java methods in the managed object. This adaptor object is named as $< ManagedObjectClassName > P2Adaptor$. This naming pattern is used by Ponder2 to load the corresponding adaptor object when requested to load a managed object. Note that it is this adaptor object that is loaded into the domain structure in place of the actual managed object. The adaptor object receives the PonderTalk messages and forwards them to the actual managed object.

Appendix C

Class Diagrams

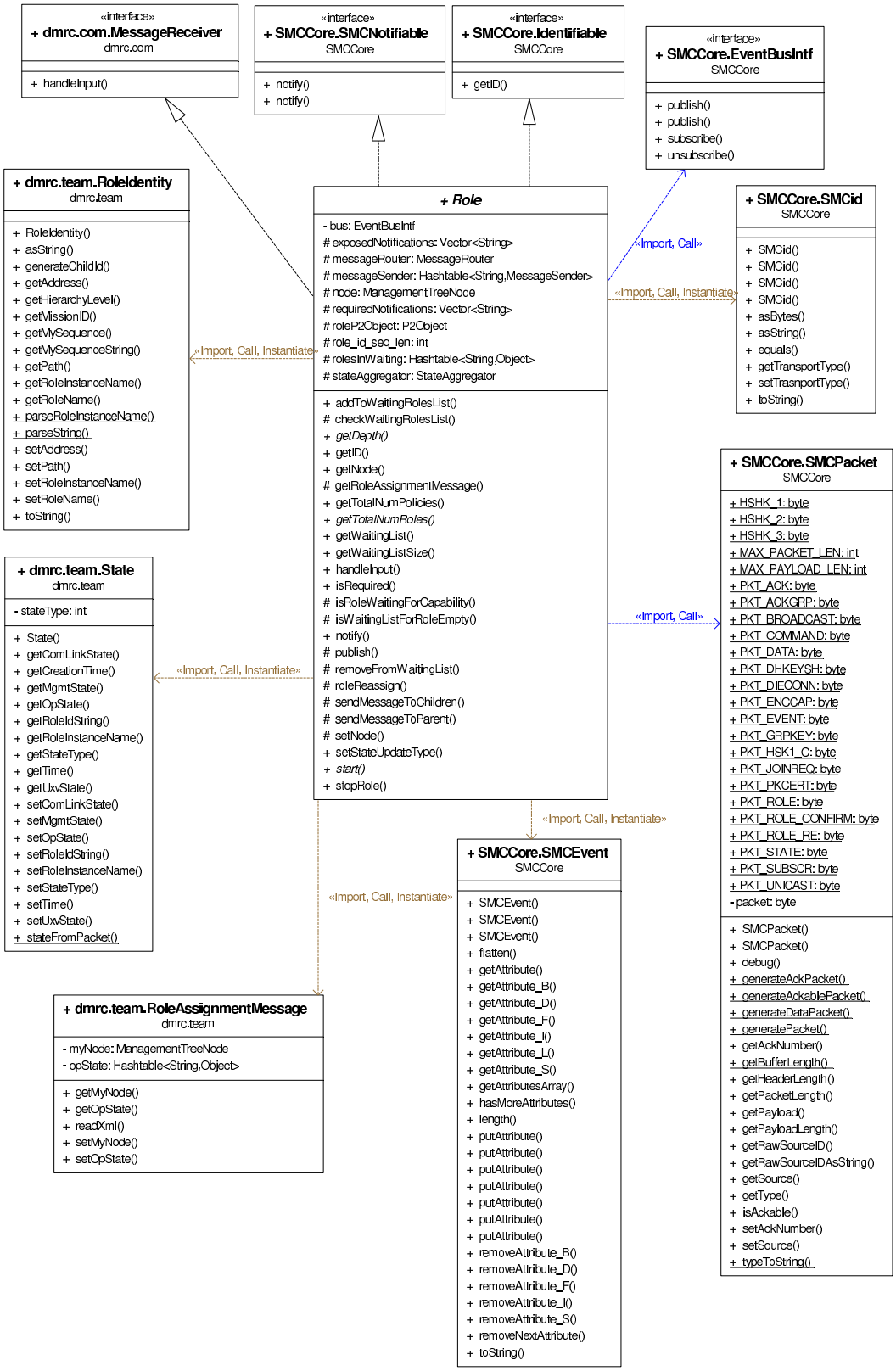


Figure C.1: The Role Class with its main Datastructures

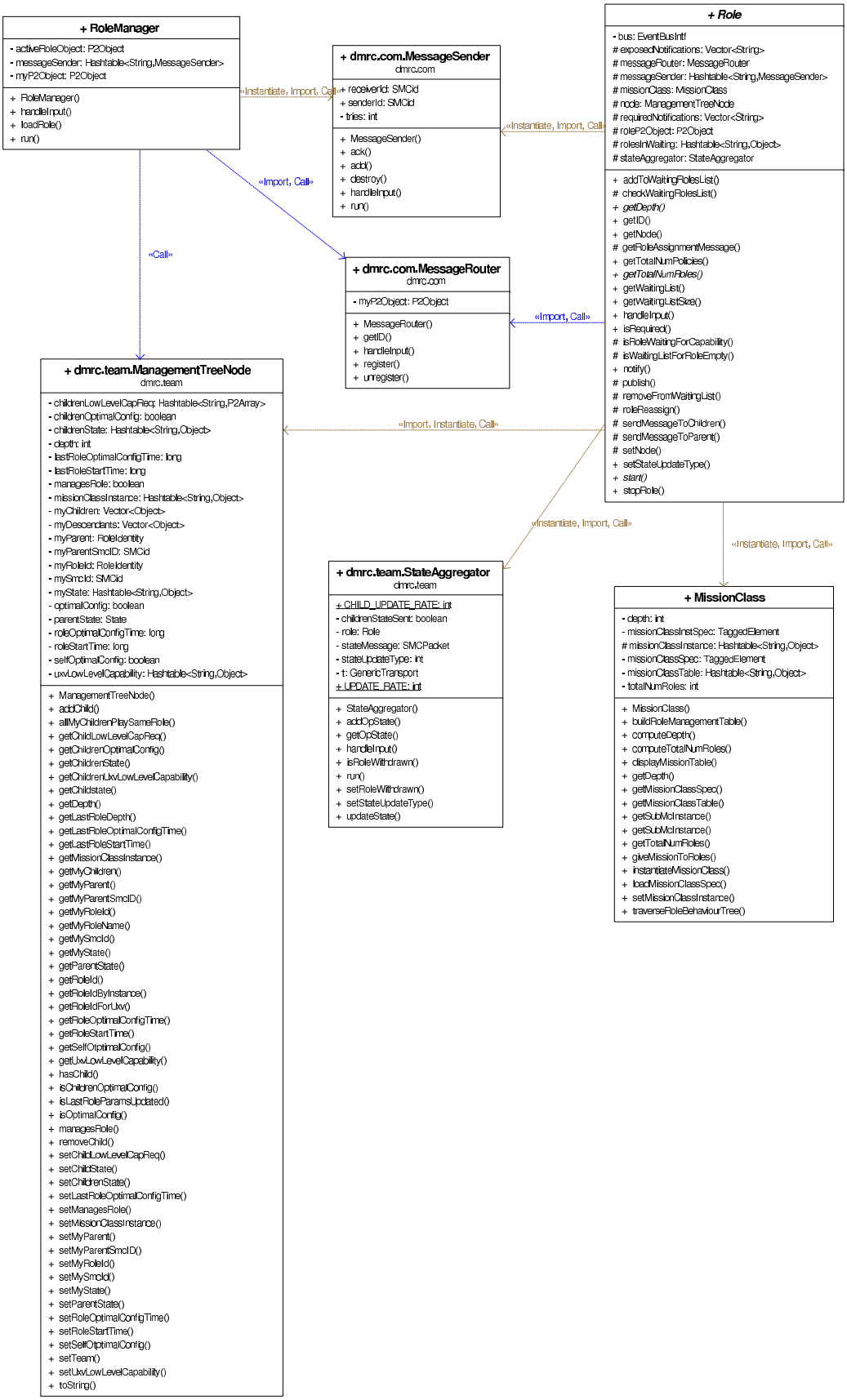


Figure C.2: Mission Layer Elements

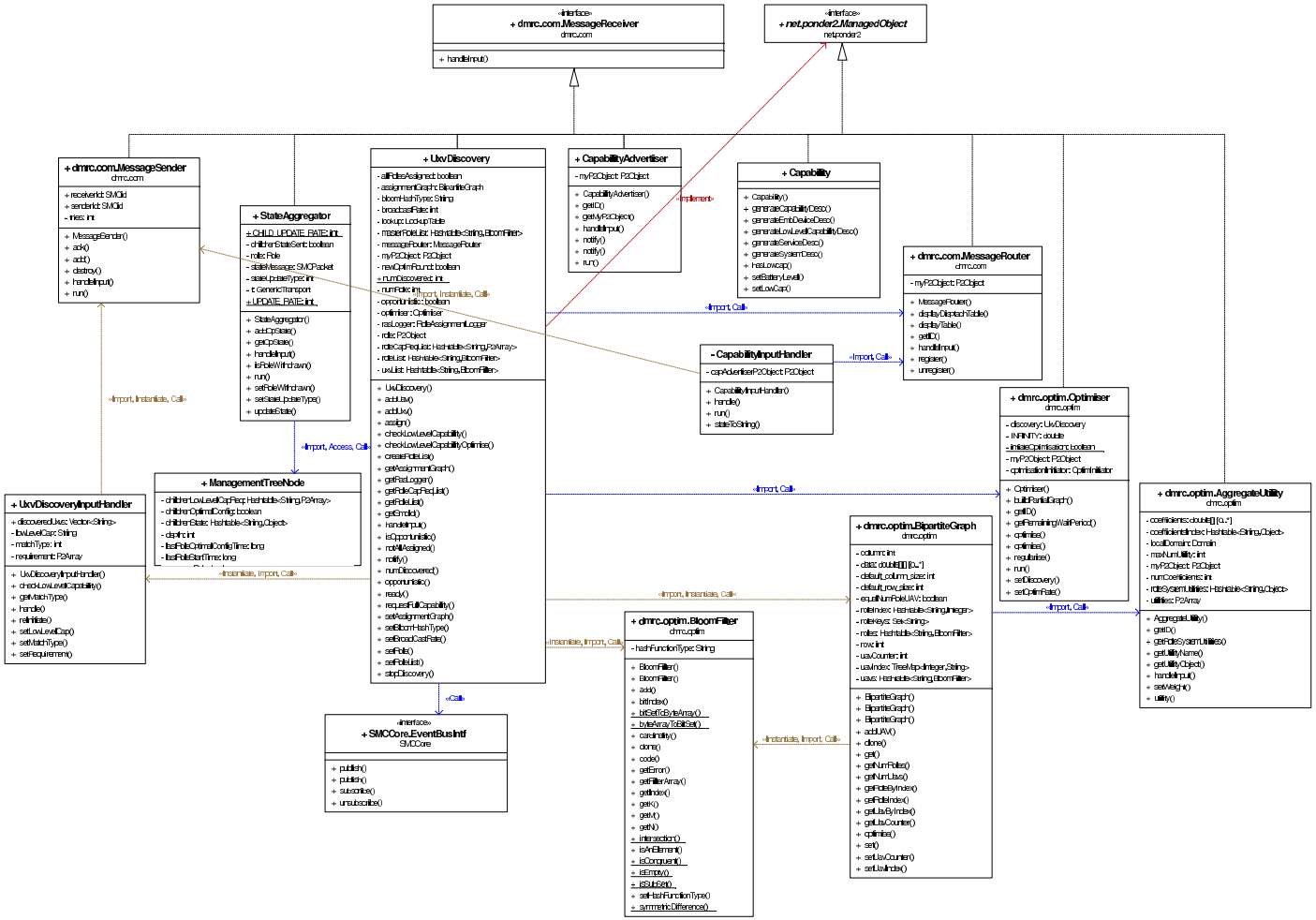


Figure C.3: Team Layer Elements

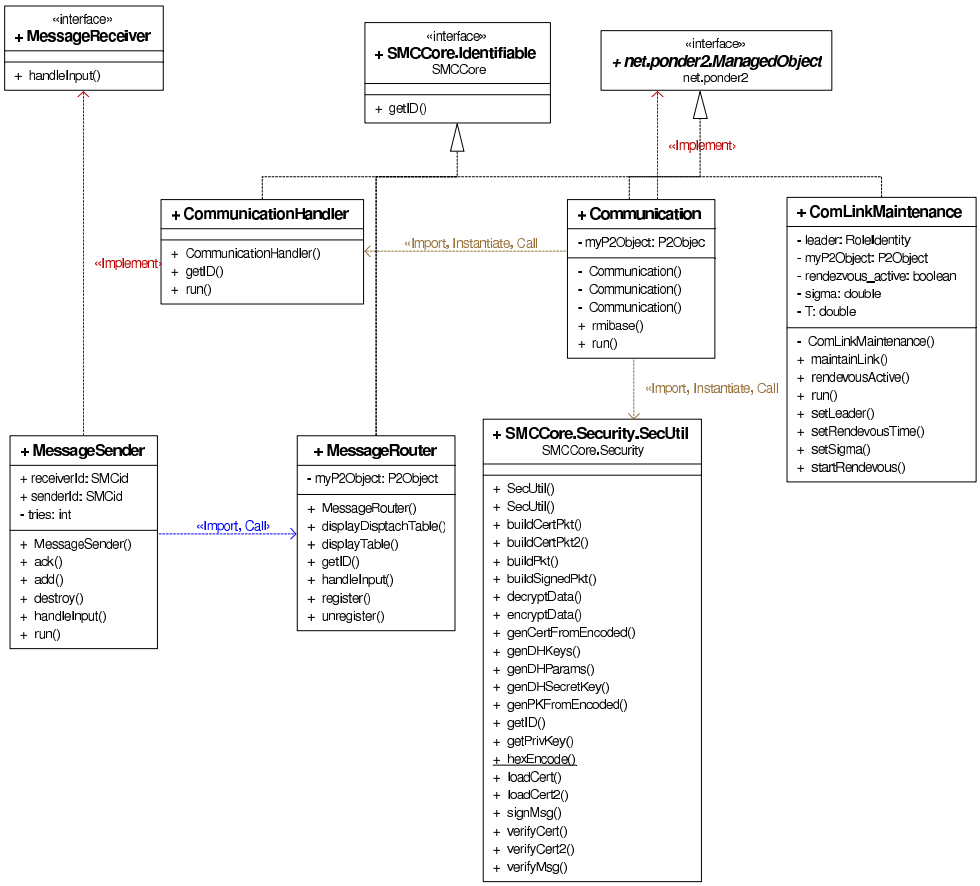


Figure C.4: Communication Layer Elements

Appendix D

Koala Robot's Low-level Control Software Interface

(a) $koa\_setPidPosition(device\_id, proportional, integral, derivative)$:sets the proportional, integral and derivative parameters of the position controller of the specified device through $device\_id$. Identifying the device is necessary because in addition to the motion controller of the robot other devices (such as camera) attached to the robot can be controlled through the KoreBot.

(b) $koa\_setPidSpeed(device\_id, proportional, integral, derivative)$: sets the proportional, integral and derivative parameters of the speed controller.

(c) $koa\_setPositionCounter(device\_id, left, right)$: sets the 32 bit position counter of the two (left and right) motors where one unit corresponds to 0.045 mm.

(d) $koa\_setPosition(device\_id, left, right)$: sets an absolute position to be reached where one unit corresponds to 0.045 mm.

(e) $koa\_readSpeed(device\_id, speed\_array[])$: reads the instantaneous left $(speed\_table[0])$ and right $(speed\_table[1])$ motor speed where one unit corresponds to 4.5mm/s.

(f) $koa\_setSpeed(device\_id, left, right)$: sets the left and right motor speed where the unit is 4.5mm/s.

(g) $koa\_readPosition(device\_id, position\_array[])$: reads the 32 bit position counter of the left $(position\_array[0])$ and right $(position\_array[1])$ motors.

(h) $koa\_setProfile(device\_id, max\_speed\_left, acceleration\_left, max\_speed\_right, acceleration\_right)$: sets the speed and the acceleration for the trapezoidal speed shape of the position controller where the units for speed and acceleration are 4.5mm/s and 1.758 mm/s^2 respectively.

(i) $koa\_readStatus(device\_id, status\_array[])$: the status is returned in three parameters, namely target ($status\_array[0]$), mode ($status\_array[1]$) and error ($status\_array[2]$). If target=0, the robot is moving, if target=1, then the robot has reached the position. If mode=0 the current displacement is controlled in the position mode and if mode =1 the current displacement is controlled in the speed mode.

(j) $koa\_readSensor(device\_id, sensor)$: reads the robot's management sensors where depending on the value of $sensor$ it provides information about : provides information about battery voltage (sensor=0), current consumption (1), ambient temperature (2) , left motor current (3), right motor current (4) and battery temperature (5).

(k) $koa\_readBattery(device\_id)$: reads the battery charge level of the robot where the unit is mAh.

(l) $koa\_readProximity(device\_id, infrared\_sensor[])$: provides the readings of 8 or 16, depending on the robot type, infra red proximity sensors.

(m) $koa\_readAmbient(device\_id, ambient\_sensor[])$: provides the readings of 8 or 16, depending on the robot type, ambient light sensors.

Bibliography

[ABH+02] A. Ankolekar, M. Burstein, J.R. Hobbs, O. Lassila, D.L. Martin, D. McDermott, S.A. McIlraith, S. Narayanan, M. Paolucci, T.R. Payne, et al. DAML-S: Web service description for the semantic web. 2002.

[AC05] Stephanie Appleyard and Victoria Chapman. Development of vignettes to be used in a study into implicit instructions and command intent in the context of autonomous systems. QinnetiQ Proprietary, Unclassified, Ref:DTC/RAO/WPE/N03571/SEAS, June 2005.

[ACI] International technology alliance in network and information science. Available at http://www.usukita.org (20/08/2008).

[ADE+00] R. Alur, A.K. Das, J.M. Esposito, R.B. Fierro, G.Z. Grudic, Y. Hur, V. Kumar, I. Lee, JP Lee, J.P. Ostrowski, et al. A Framework and Architecture for Multirobot Coordination. *Lecture Notes in Control and Information Sciences; Vol. 271*, pages 303–312, 2000.

[AGH+00] Rajeev Alur, Radu Grosu, Yerang Hur, Vijay Kumar, and Insup Lee. Modular specification of hybrid systems in charon. In *HSCC '00: Proceedings of the Third International Workshop on Hybrid Systems: Computation and Control*, pages 6–19, London, UK, 2000. Springer-Verlag.

[AGS+09] E. Asmare, A. Gopalan, M. Sloman, N. Dulay, and E. Lupu. A Mission Management Framework for Unmanned Autonomous Vehicles. In *Mobile Wireless Middleware: Operating Systems and Applications. Second International Conference, Mobilware 2009, Berlin, Germany, April 28-29, 2009. Proceedings*, page 222. Springer, 2009.

[AHW03] Naveed Arshad, Dennis Heimbigner, and Alexander. L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. In *ICTAI '03: Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*, page 39, Washington, DC, USA, 2003. IEEE Computer Society.

[AHW07] N. Arshad, D. Heimbigner, and A.L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. *Software Quality Journal*, 15(3):265–281, 2007.

[Alb93] J.S. Albus. A reference model architecture for intelligent systems design. 1993.

[And02] F. Andreasen. Session description protocol (SDP) simple capability declaration. Request for comments 3407, The Internet Society, 2002.

[AOSY99] H. Ando, Y. Oasa, I. Suzuki, and M. Yamashita. Distributed memoryless point convergence algorithm for mobile robots with limited visibility. *IEEE Transactions on Robotics and Automation*, 15(5):818–828, October 1999.

[Ark87] R. Arkin. Motor schema based navigation for a mobile robot: An approach to programming by behavior. *Robotics and Automation. Proceedings. 1987 IEEE International Conference on*, 4, 1987.

[Ark98] Ronald C. Arkin. *A Behavior-based Robotics*. MIT Press, Cambridge, MA, USA, 1998.

[BAFH84] A.J. Barbera., J.S. Albus, M.L. Fitzgerald, and L.S. Haynes. RCS: The NBS real-time control system. In *Robots 8 Conference and Exposition*, Detroit, MI, June 1984.

[Bek05] George A. Bekey. *Autonomous Robots: From Biological Inspiration to Implementation and Control*. The MIT Press, first edition, 2005.

[BFG+97] R.P. Bonasso, R.J. Firby, E. Gat, D. Kortenkamp, D.P. Miller, and M.G. Slack. Experiences with an architecture for intelligent, reactive agents. *JETAI*, 9(2-3):237–256, 1997.

[BL71] François Bourgeois and Jean-Claude Lassalle. An extension of the munkres algorithm for the assignment problem to rectangular matrices. *Commun. ACM*, 14(12):802–804, 1971.

[BLMR04] A.K. Bandara, E.C. Lupu, J. Moffett, and A. Russo. A goal-based approach to policy refinement. In *Proceedings of the 5th IEEE Workshop on Policies for Distributed Systems and Networks*, 2004.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[BLR+06] Arosha K. Bandara, Emil C. Lupu, Alessandra Russo, Naranker Dulay, Morris Sloman, Paris Flegkas, Marinos Charalambides, and George Pavlou. Policy refinement for ip differentiated services quality of service management. *Network and Service Management, IEEE Transactions on*, 3(2):2–13, April 2006.

[BM04] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.

[BMY02] Jean Bacon, Ken Moody, and Walt Yao. A model of oasis role-based access control and its support for active security. *ACM Trans. Inf. Syst. Secur.*, 5(4):492–540, 2002.

[Bri] Encyclopdia Britannica. autonomic nervous system. Available at http://www.britannica.com/EBchecked/topic/45079/autonomic-nervous-system (20/08/2009).

[Bro86] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[Bro90] R.A. Brooks. The Behavior Language; User's Guide. Technical report, MIT, 1990.

[Bur93] M. Burgess. Cfengine: a system configuration engine. Technical report, University of Oslo, 1993.

[Bur95] M. Burgess. Cfengine: a site configuration engine. *USENIX Computing systems*, 8(3), 1995. http://www.cfengine.org/pages/science.

[CBCD91] L. Champeny-Bares, S. Coppersmith, and K. Dowling. *The terregator mobile robot (Technical Report: CMU-RI-TR-93-03)*. Carnegie Mellon University, The Robotics Institute, 1991.

[CCJ90] Brent N. Clark, Charles J. Colbourn, and David S. Johnson. Unit disk graphs. *Discrete Mathematics*, 86(1-3):165–177, 1990.

[CGM+04] R. Chinnici, M. Gudgin, J.J. Moreau, J. Schlimmer, and S. Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. *W3C Working Draft*, 26, 2004.

[CKC04] Luiz Chaimowicz, Vijay Kumar, and Mario F. M. Campos. A paradigm for dynamic coordination of multiple robots. *Auton. Robots*, 17(1):7–21, 2004.

[CL91] P.R. Cohen and H.J. Levesque. Teamwork. *Nous*, pages 487–512, 1991.

[CLM+09] Robert Craven, Jorge Lobo, Jiefei Ma, Alessandra Russo, Emil Lupu, and Arosha Bandara. Expressive policy analysis with enhanced system dynamicity. In *ASIACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 239–250, New York, NY, USA, 2009. ACM.

[CM05] J. Carlson and RR Murphy. How UGVs physically fail in the field. *IEEE Transactions on Robotics*, 21(3):423–437, 2005.

[CMT88] G. Carpaneto, S. Martello, and P. Toth. Algorithms and codes for the assignment problem. *Annals of Operations Research*, 13(1):191–223, 1988.

[CSWW04] D.M. Chess, A. Segal, I. Whalley, and S.R. White. Unity: Experiences with a prototype autonomic computing system. In *Proceedings of the First International Conference on Autonomic Computing*, pages 140–147. IEEE Computer Society, 2004.

[Dam02] Nicodemos C. Damianou. *A Policy Framework for Management of Distributed Systems*. phd thesis, Imperial College of Science, Technology and Medicine, February 2002.

[DDLS01] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder policy specification language. In *Proceedings of Policy 2001, Workshop on Policies for Distributed Systems and Networks*, pages 18–39. Springer-Verlag LNCS, 2001.

[DH76] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[Dic] Merriam-Webster Online Dictionary. Mission. Available at http://www.merriam-webster.com/dictionary/mission (20/08/2009).

[DKFS02] Mads Dam, Gunnar Karlsson, Babak Sadighi Firozabadi, and Rolf Stadler. A research agenda for distributed policy-based management. Report, The Royal Institute of Technology(KTH), 2002.

[DLSD01] N. Dulay, E. Lupu, M. Sloman, and N Damianou. A policy deployment model for the ponder language. In *Proceedings of IEEE/IFIP International Symposium on Intergrated Network Management*, pages 529–543. IEEE press, May 2001.

[DML03] S.A. DeLoach, E.T. Matson, and Y. Li. Exploiting agent oriented software engineering in cooperative robotics search and rescue. *International Journal of Pattern Recognition and Artificial Intelligence*, 17(5):817–835, 2003.

[DMT03] DMTF. CIM concepts white paper. White paper CIM Versions 2.4+, Document Version 0.9, Distributed Management Task Force, Inc.(DMTF), June 2003.

[DS83] R. Davis and R.G. Smith. Negotiation as a Metaphor for Distributed Problem Solving. *Artificial Intelligence*, 20:63–109, 1983.

[DVSD04] V. Dignum, J. Vazquez-Salceda, and F. Dignum. Omni: Introducing social structure, norms and ontologies into agent organizations. In *ProMAS*, volume 2004, pages 181–198. Springer, 2004.

[DWS01] S.A. DeLoach, M.F. Wood, and C.H. Sparkman. Multiagent systems engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):231–258, 2001.

[FFMM94] Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. Kqml as an agent communication language. In *CIKM '94: Proceedings of the third international conference on Information and knowledge management*, pages 456–463, New York, NY, USA, 1994. ACM.

[GM01] Brian P. Gerkey and Maja J. Mataric. Principled communication for dynamic multi-robot task allocation. In *ISER '00: Experimental Robotics VII*, pages 353–362, London, UK, 2001. Springer-Verlag.

[GM02] BP Gerkey and MJ Mataric. Sold!: Auction methods for multirobot coordination. *IEEE Transactions on Robotics and Automation*, 18(5):758–768, 2002.

[H+99] R. Housley et al. Internet X. 509 Public Key Infrastructure Certificate and CRL Profile. Technical report, RFC 2459, January, 1999.

[Hab07] M.K. Habib. Humanitarian Demining: Reality and the Challenge of Technology-The State of the Arts. *International Journal of Advanced Robotic Systems*, 4(2):151–172, 2007.

[Hen96] T.A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS*, 1996.

[HJS06] T.D. Huynh, N.R. Jennings, and N.R. Shadbolt. An integrated trust and reputation model for open multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 13(2):119–154, 2006.

[HK73] J.E. Hopcroft and R.M. Karp. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM Journal on Computing*, 2:225, 1973.

[HM00] J. Hendler and D.L. McGuinness. The DARPA agent markup language. *IEEE Intelligent systems*, 15(6):67–73, 2000.

[Hor01] P. Horn. Autonomic computing: IBMs perspective on the state of information technology. IBM Corporation, October 2001.

[IEE07] Ieee standard for information technology-telecommunications and information exchange between systems-local and metropolitan area networks-specific requirements - part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. *IEEE Std 802.11-2007 (Revision of IEEE Std 802.11-1999)*, pages C1–1184, 12 2007.

[IHAK02] A.A. Ivan, J. Harman, M. Allen, and V. Karamcheti. Partitionable services: A framework for seamlessly adapting distributed applications to heterogeneous environments. In *11th IEEE International Symposium on High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings*, pages 103–112, 2002.

[IMN04] M. Ionescu, N. Minsky, and T. Nguyen. Enforcement of communal policies for peer-to-peer systems. In *Proc. of the Sixth International Conference on Coordination Models and Languages*. Citeseer, 2004.

[Inc] C.O. Inc. CPLEX Linear Optimizer and Mixed Integer Optimizer. *Suite*, 279:930.

[INPS03] Luca Iocchi, Daniele Nardi, Maurizio Piaggio, and Antonio Sgorbissa. Distributed coordination in heterogeneous multi-robot systems. *Auton. Robots*, 15(2):155–168, 2003.

[Jen00] Nicholas R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2):277–296, 2000.

[Jet] Jet Propulsion Labratory. Mars exploration rover mission. Available at http://marsrovers.nasa.gov/overview/ (20/08/2009).

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[KIK03] T. Kichkaylo, A. Ivan, and V. Karamcheti. Constrained component deployment in wide-area networks using AI planning techniques. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 10–, 2003.

[KNS06] M. Koes, I. Nourbakhsh, and K. Sycara. Constraint optimization coordination architecture for search and rescue robotics. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 3977–3982, May 2006.

[Kon97] K. Konologie. The Saphira architecture: A design for autonomy. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2):215–235, 1997.

[kt] k team. Available at http://www.k-team.com (20/08/2009).

[Kuh55] HW Kuhn. THE HUNGARIAN METHOD FOR THE ASSIGNMENT PROBLEM1. *Naval research logistics quarterly*, page 83, 1955.

[Kuh56] H.W. Kuhn. Variants of the Hungarian method for assignment problems. *Naval Research Logistics Quarterly*, 3:253–258, 1956.

[KW04] JO Kephart and WE Walsh. An artificial intelligence perspective on autonomic computing policies. *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, pages 3–12, 2004.

[LA89] DM Lyons and MA Arbib. A formal model of computation for sensory-based robotics. *Robotics and Automation, IEEE Transactions on*, 5(3):280–293, 1989.

[LCN90] H.J. Levesque, P.R. Cohen, and J.H.T. Nunes. On acting together. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, pages 94–99. Boston, MA, 1990.

[LDS+08] E. Lupu, N. Dulay, M. Sloman, J. Sventek, S. Heeps, S. Strowes, K. Twidle, S.L. Keoh, and A. Schaeffer-Filho. AMUSE: Autonomic management of ubiquitous e-health systems. *Concurrency and Computation: Practice & Experience*, 20(3):277–295, 2008.

[Lin05] Jie Lin. Distributed mobility control for fault-tolerant mobile networks. *In Proceedings of Systems Communications*, 2005.

[LMA07] J. Lin, A. S. Morse, and B. D. O. Anderson. The multi-agent rendezvous problem. part 1: The synchronous case. *SIAM J. Control Optim.*, 46(6):2096–2119, 2007.

[LMSY96] Emil C. Lupu, Damian A. Marriott, Morris S. Sloman, and Nicholas Yialelis. A policy based role framework for access control. In *RBAC '95: Proceedings of the first ACM Workshop on Role-based access control*, page 11, New York, NY, USA, 1996. ACM.

[LMW02] N. Li, JC Mitchell, and WH Winsborough. Design of a role-based trust-management framework. In *2002 IEEE Symposium on Security and Privacy, 2002. Proceedings*, pages 114–130, 2002.

[LOC00] M. Lindström, A. Orebäck, and HI Christensen. BERRA: A research architecture for service robots. *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, 4:3278–3283, 2000.

[LS99] EC Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on software engineering*, 25(6):852–869, 1999.

[Ltd] Cyberbotics Ltd. Webots. Available at http://www.cyberbotics.com (20/08/09).

[M+01] B. Moore et al. Policy core information model. Request for Comments 3060 Version 1, The Internet Society, February 2001.

[MA92] RR Murphy and RC Arkin. SFX: An architecture for action-oriented sensor fusion. *Intelligent Robots and Systems, 1992., Proceedings of the 1992 lEEE/RSJ International Conference on*, 2, 1992.

[MAC97] Douglas C. MacKenzie, Ronald C. Arkin, and Jonathan M. Cameron. Multiagent mission specification and execution. *Auton. Robots*, 4(1):29–52, 1997.

[Mey93] Alex Meystel. *An Introduction to Intelligent and Autonomous Control*, chapter Nested Hierarchical Control, pages 129–161. Kluwer Academic Publishers, 1993.

[MFO+06] A. Morris, D. Ferguson, Z. Omohundro, D. Bradley, D. Silver, C. Baker, S. Thayer, C. Whittaker, and W. Whittaker. Recent developments in subterranean robotics. *Journal of Field Robotics*, 23(1), 2006.

[MMRM05] S. Malek, M. Mikic-Rakic, and N. Medvidovic. A decentralized redeployment algorithm for improving the availability of distributed systems. *Lecture notes in computer science*, 3798:99, 2005.

[MPB08] Robin R. Murphy, Kevin S. Pratt, and Jennifer L. Burke. Crew roles and operational protocols for rotary-wing micro-uavs in close urban environments. In *HRI '08: Proceedings of the 3rd ACM/IEEE international conference on Human robot interaction*, pages 73–80, New York, NY, USA, 2008. ACM.

[MRMM05] M. Mikic-Rakic, S. Malek, and N. Medvidovic. Improving availability in large, distributed component-based systems via redeployment. *Lecture notes in computer science*, 3798:83, 2005.

[MS94] J.D. Moffett and M.S. Sloman. Policy conflict analysis in distributed system management. *Journal of Organizational Computing*, 4:1–22, 1994.

[MSG+08] Robin R. Murphy, Eric Steimle, Chandler Griffin, Charlie Cullins, Mike Hall, and Kevin Pratt. Cooperative use of unmanned sea surface and micro aerial vehicles at hurricane wilma. *J. Field Robot.*, 25(3):164–180, 2008.

[MU00] Naftaly H. Minsky and Victoria Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Trans. Softw. Eng. Methodol.*, 9(3):273–305, 2000.

[Mun57] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[Mur00] Robin R. Murphy. *Introduction to AI Robotics*. MIT Press, Cambridge, MA, USA, 2000.

[Mur04a] Robin R. Murphy. Rescue robotics for homeland security. *Commun. ACM*, 47(3):66–68, 2004.

[Mur04b] R.R. Murphy. Trial by fire. *IEEE robotics & automation magazine*, 11(3):50–61, 2004.

[Nil94] Nils J. Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.

[NR03] P. Norvig and SJ Russell. *Artificial intelligence: a modern approach*. Prentice Hall, 2003.

[OAS03] OASIS Provisioning Services Technical Committee. Service provisioning markup language (SPML), June 2003.

[Omi01] A. Omicini. SODA: Societies and infrastructures in the analysis and design of agent-based systems. *Lecture Notes in Computer Science*, pages 185–194, 2001.

[Par94] L.E. Parker. Heterogeneous multi-robot cooperation. 1994.

[Par96] L.E. Parker. L-ALLIANCE: Task-oriented multi-robot learning in behavior-based systems. *Advanced Robotics*, 11(4):305–322, 1996.

[Par98] LE Parker. ALLIANCE: An architecture for fault tolerant multirobot cooperation. *Robotics and Automation, IEEE Transactions on*, 14(2):220–240, 1998.

[Per98] C. E. Perkins. Mobile ad hoc networking terminology. INTERNET DRAFT Version 1, Internet Engineering Task Force, November 1998.

[PH05] Manish Parashar and Salim Hariri. Autonomic computing: An overview. *Lecture Notes in Computer Science*, 3566:257–269, August 2005.

[Phi07] D. Philpott. Border Security: New Eyes in the Skies. *Homeland Defense Journal*, 5(2):5, 2007.

[PKPS02] M. Paolucci, T. Kawamura, T.R. Payne, and K. Sycara. Semantic matching of web services capabilities. *Lecture Notes in Computer Science*, pages 333–347, 2002.

[Pon] Ponder2. Available at http://ponder2.net (20/08/2009).

[RHJ05] S.D. Ramchurn, D. Huynh, and N.R. Jennings. Trust in multi-agent systems. *The Knowledge Engineering Review*, 19(01):1–25, 2005.

[RKM+07] K.A. Remley, G. Koepke, E. Messina, A. Jacoff, and G. Hough. Standards development for wireless communications for urban search and rescue robots. In *9th Ann. Int'l Symp. on Advanced Radio Tech*, pages 26–28, 2007.

[RLH06] Y. Rekhter, T. Li, and S. Hares. RFC 4271: a Border Gateway Protocol 4 (BGP-4). Request for comments 4271, The Internet Society, January 2006.

[Sch05] C. Schlenoff. A robot ontology for urban search and rescue. In *Proceedings of the 2005 ACM workshop on Research in knowledge representation for autonomous systems*, pages 27–34. ACM New York, NY, USA, 2005.

[She71] William Shelton. The united states and the soviet union: Fourteen years in space. *Russian Review*, 30(4):322–334, 1971.

[SKWL99] K. Sycara, M. Klusch, S. Widoff, and J. Lu. Dynamic service matchmaking among agents in open information environments. *ACM SIGMOD Record*, 28(1):47–53, 1999.

[SLK98] K. Sycara, J. Lu, and M. Klusch. Interoperability among heterogeneous software agents on the Internet. *The Robotics Institute, Carnegie Mellon University, Pittsburgh, USA*, page 35, 1998.

[Slo94] M. Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2(4):333–360, 1994.

[Smi80] R.G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *Computers, IEEE Transactions on*, C-29(12), 1980.

[SN00] National Search and Rescue Committee (NSRC). United States National Search and Rescue Supplement to the International Aeronautical and Maritime Search and Rescue Manual. 2000.

[SS] Wei-Min Shen and B. Salemi. Distributed and dynamic task reallocation in robot organizations. *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on*, 1.

[SWKL02] K. Sycara, S. Widoff, M. Klusch, and J. Lu. Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace. *Autonomous agents and multi-agent systems*, 5(2):173–203, 2002.

[SY99] I. Suzuki and M. Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999.

[Tam97] M. Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence*, 7:83–124, 1997.

[TCF+02] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, and P. Vanderbilt. Grid service specification. In *Open Grid Service Infrastructure WG, Global Grid Forum, Draft*, volume 2, page 17, 2002.

[TCW+04] Gerald Tesauro, David M. Chess, William E. Walsh, Rajarshi Das, Alla Segal, Ian Whalley, Jeffrey O. Kephart, and Steve R. White. A multi-agent systems approach to autonomic computing. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 464–471. IEEE Computer Society, 2004.

[THRR06] WF Truszkowski, MG Hinchey, JL Rash, and CA Rouff. Autonomous and autonomic systems: A paradigm for future space exploration missions. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 36(3):279–291, 2006.

[Tid93] G. Tidhar. Team-oriented programming: Preliminary report. *Technical Note*, 41, 1993.

[TPC00] M. Tambe, DV Pynadath, and N. Chauvat. Building dynamic agent organizations in cyberspace. *IEEE Internet Computing*, 4(2):65–73, 2000.

[UEWA07] P. Ulam, Y. Endo, A. Wagner, and R. Arkin. Integrated mission specification and task allocation for robot teams - design and implementation. *IEEE International Conference on Robotics and Automation*, pages 4428–4435, April 2007.

[Upna] UpnP Forum. Upnp. Available at http://www.upnp.org (20/08/2009).

[Upnb] UpnP Forum. UpnP Resources. Available at http://www.upnp.org/resources/upnpresources20051215.zip (20/08/2009).

[Ver02] Dinesh C. Verma. Simplifying network administration using policy-based management. *IEEE Network*, 16(2):20–26, March/April 2002.

[VSD03] Javier Vázquez-Salceda and Frank Dignum. Modelling electronic organizations. In *CEEMAS'03: Proceedings of the 3rd Central and Eastern European conference on Multi-agent systems*, pages 584–593, Berlin, Heidelberg, 2003. Springer-Verlag.

[VSDF05] J. Vzquez-Salceda, V. Dignum, and F.Dignum. Organizing multiagent systems. *Autonomous Agents and Multi-Agent System*, 11(3):307–360, November 2005.

[W+01] A. Westerinen et al. Terminology for policy-based management. Request for comments 3198, The Internet Society, November 2001.

[Wer00] B.B. Werger. Ayllu: Distributed port-arbitrated behavior-based control. In *Proceedings, The 5th Intl. Symp. on Distributed Autonomous Robotic Systems*, pages 25–34, 2000.

[WHW+04] Steve R. White, James E. Hanson, Ian Whalley, David M. chess, and Jeffrey O. Kephart. An architectural approach to autonomic computing. In *Proceedings of the International Conference on Autonomic Computing*, pages 2–9. IEEE Computer Society, May 2004.

[WJK00] M. Wooldridge, N.R. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[WM00] B.B. Werger and M.J. Mataric. Broadcast of local eligibility for multi-target observation. In *Proceedings, 5th International Symposium on Distributed Autonomous Robotic Systems (DARS)*, 2000.

[WN98] B. Wilcox and T. Nguyen. Sojourner on Mars and lessons learned for future planetary rovers. In *SAE, International Conference on Environmental Systems, 28 th, Danvers, MA*, 1998.

[WPT03] R. Want, T. Pering, and D. Tennenhouse. Comparing autonomic and proactive computing. *IBM Syst. J.*, 42(1):129–135, 2003.

[WR+04] J. Wong, C. Robinson, et al. Urban Search and Rescue Technology Needs: Identification of Needs. *Department of Homeland Security/FEMA, Final report*, 1, 2004.

[Yam04] B. Yamauchi. PackBot: A versatile platform for military robotics. *Proceedings of SPIE Conference 5422: Unmanned Ground Vehicles VI*, 2004.