

Maintenance Goals in Intelligent Agents

A thesis submitted in fulfilment of the requirements for the degree of
Master of Computer Science

Simon Alan Duff

B.Eng. Computer Systems (Honours),
B.App.Sci. Computer Science (Honours),

School of Computer Science and Information Technology,
College of Science, Engineering and Health,
RMIT University,
Melbourne, Victoria, Australia.

August, 2009

Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; any editorial work, paid or unpaid, carried out by a third party is acknowledged; and, ethics procedures and guidelines have been followed.

Simon Alan Duff

February 8, 2010

Acknowledgments

Firstly, I would like to thank my supervisors, James Harland and John Thangarajah, for their support, not only with my research and thesis, but also in other events in my life. They have acted in a manner above and beyond what is expected of them as supervisors, and for that, I thank them sincerely.

I would like to thank several of my fellow research students, especially Chris, Jenn, David, Dennis and Antony. It was of great help to have such kind friends to support me and to socialise with throughout this (sometimes trying) process.

The RMIT Agent group was also beneficial in my research, allowing me to engage in frequent discussions of my work, which only helped to improve the ideas and presentation found in this thesis.

Thanks also to the Computer Science ‘Technical Services Group’ for their hospitality, and for ensuring that servers and machines were functional.

Last, but by no means least, I would like to thank my wife Shereene and my family and friends for their love, motivation and guidance through out my entire academic career. This thesis is a testament to them and their support, as much as it is to me.

Thank you one and all.

Credits

Portions of the material in this thesis have previously appeared in the following publications:

- S. Duff, J. Harland and J Thangarajah. On Proactivity and Maintenance Goals. *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multi-Agent Systems*, 1033–1040, 2006
- S. Duff and J. Harland. Formalising Maintenance Goals. *Declarative Agent Languages and Technologies*, 157–172, 2008

The thesis was written in the `Vim` editor on Ubuntu GNU/Linux, and typeset using the \LaTeX 2 ϵ document preparation system. Experimental work was performed using Sun Java. Results were analysed using `Python`, and graphs generated using `matplotlib`. Miscellaneous diagrams were generated using `XFig`.

All trademarks are the property of their respective owners.

Note

Unless otherwise stated, all fractional results have been rounded to the displayed number of decimal figures.

Contents

Abstract	1
1 Introduction	2
1.1 Contributions	4
2 Background	8
2.1 Agents and Goals	8
2.1.1 Agent Architectures	9
Reactive Agents	10
Hybrid Agents	12
2.2 BDI	14
2.3 Goals	18
2.3.1 Achievement Goals	18
2.3.2 Maintenance Goals	19
2.3.3 Guarded Actions	19
2.3.4 Reactive Maintenance Goals	20
2.3.5 Proactive Maintenance Goals	21
2.3.6 Alternate Approaches	22
2.3.7 Other Goal Types	24
2.3.8 Properties of Goals	24
2.4 Plans	26
2.4.1 Goal-Plan Trees	26
2.5 Goal Interaction	27
2.5.1 Interaction due to resources	28
2.5.2 Interaction due to effects	28

2.6	Maintenance goals in practical agent systems	29
2.6.1	PRS, dMARS, JACK	29
2.6.2	JAM	30
2.6.3	Jadex	30
2.6.4	AgentSpeak and Jason	32
2.6.5	3APL, GOAL and Dribble	33
3	Representation and Reasoning	35
3.1	Case Studies	35
3.1.1	Mars Rover	35
	Maintenance Goals with the Mars Rover	36
	Maintenance Goals with Proactive behaviour	37
3.1.2	Robot Soccer	39
3.2	Representation	41
3.2.1	Achievement Goals	41
3.2.2	Plans	42
3.2.3	Maintenance Goals	43
3.3	Algorithms	44
3.3.1	Reactive Behaviour	45
3.3.2	Proactive Behaviour	45
3.3.3	Mars Rover Revisited	48
	Reactive Behaviour Only	50
	Reactive and Proactive Behaviour	50
3.3.4	Potential Optimisations	51
4	Operational Semantics	53
4.1	Issues for Formalisation	53
4.2	Maintenance Goal Formalisation	54
4.2.1	Operation	55
4.3	Complete State Transition Rules	56
4.4	The future Operator	60
4.4.1	Resource Summary based	60
	Cautious vs Bold Agents	60
4.4.2	User Supplied	61

4.5	Case Studies	62
4.5.1	Example of Maintenance Goals with proactive behaviour	64
5	Experimental Results	68
5.1	Experimental Overview	68
5.1.1	Experimental Setup	69
5.1.2	Definitions and Terminology	71
	Measurement of each trial	71
	Outcomes of trials	71
5.2	Hypothesis	73
5.3	Results	73
5.3.1	Maintenance goals in error-free environments	74
5.3.2	Varying errors in Maintenance goals	74
5.3.3	Overestimation in Maintenance goals	77
5.3.4	Underestimation in Maintenance goals	79
6	Conclusion	82
	Bibliography	86

List of Figures

2.1	Example of a simple Subsumption Architecture based agent	10
2.2	TouringMachine Architecture	12
2.3	InterRRap Architecture	13
2.4	Abstract Agent Interpreter	17
2.5	An example of a goal-plan tree	27
2.6	Example Goals in JAM	30
2.7	Maintenance Goal Life-cycle in Jadex (<i>Taken from Pokahr et al. [2003]</i>) . . .	31
2.8	Example Goals in Jason	33
3.1	Scenario Overview	38
3.2	Step-by-Step Reactive Agent Example	39
3.3	Step-by-Step Proactive-behaviour Agent Example	40
3.4	Scenario with reactive behaviour	40
3.5	Scenario with proactive behaviour	40
3.6	Achievement Goal Definition	42
3.7	Plan Definition	43
3.8	Maintenance Goal Definition	43
3.9	The proactive-check algorithm	47
3.10	MoveTo20Goal Specifications	48
3.11	Example Reactive Maintenance Goal Specification	49
3.12	Example Maintenance Goal Specification utilising proactive-check	50
4.1	Goal Life-cycle from van Riemsdijk et al. [2008]	56
4.2	Formal Semantics from van Riemsdijk et al.'s framework	57
4.3	Modified Operational Semantics	58

5.1	Example locations	69
5.2	Categorisation of movement types	72
5.3	10 locations, fuel tank of 100	75
5.4	10 locations, fuel tank of 200	75
5.5	10000 locations, fuel tank of 100	75
5.6	10000 locations, fuel tank of 200	75
5.7	10 locations, fuel tank of 100	76
5.8	10 locations, fuel tank of 200	76
5.9	10000 locations, fuel tank of 100	76
5.10	10000 locations, fuel tank of 200	76
5.11	10 locations, fuel tank of 100	78
5.12	10 locations, fuel tank of 200	78
5.13	10000 locations, fuel tank of 100	78
5.14	10000 locations, fuel tank of 200	78
5.15	10 locations, fuel tank of 100	80
5.16	10 locations, fuel tank of 200	80
5.17	10000 locations, fuel tank of 100	80
5.18	10000 locations, fuel tank of 200	80

Abstract

One popular software development strategy is that of intelligent agent systems. Agents are often programmed by goals; a programmer or user defines a set of goals for an agent, and then the agent is left to determine how best to complete the goals assigned to them. Popular types of goals are achievement and maintenance goals.

An achievement goal describes some particular state the agent would like to bring about, for example, being in a particular location or having a particular bank balance. Given an achievement goal, an agent will perform actions that it believes will lead it to having the achievement goal realised.

In current agent systems, maintenance goals tell an agent to ensure that some condition is always kept satisfied, for example, ensuring that a vehicle stays below a certain speed, or that it has sufficient fuel in its fuel tank. Currently, maintenance goals are *reactive*, in that they are not considered until *after* the maintenance condition has been violated. Only then does the agent begin to perform actions to restore the maintenance condition.

In this thesis, we have discussed methods by which maintenance goals can be made *proactive*. Proactive maintenance goals may cause an agent to perform actions *before* a maintenance condition is violated, when it can *predict* that a maintenance condition will be violated in the future. This can be due to changes to the environment, or more interestingly, when the agent itself is performing actions that will cause the violation of the maintenance condition. Operational semantics that clearly demonstrate the functionality and operation of proactive maintenance goals have been developed in this thesis. We have experimentally shown that maintenance goals with proactive behaviour will reduce the amount of resources consumed in a variety of error-prone environments. This includes scenarios where the agent's beliefs are less than the true values, as well as when the beliefs are in excess of the true values.

Chapter 1

Introduction

Intelligent agents are gaining acceptance as a practical solution to many computing problems that require timely and goal-directed behaviour in highly dynamic domains. Such scenarios include air traffic control[Ljungberg and Lucas, 1992], on-board spacecraft diagnosis[Muscettola et al., 1998] and entertainment such as games[Evans, 2002].

Intelligent agents have been found suitable in such environments and scenarios due to their autonomy, which enables them to perform tasks without continual intervention by a human operator. This makes agents suitable for situations when a human may not be available to guide it (such as in a spacecraft scenario), or when a decision must be made quickly – this describes environments that are both highly dynamic and complex.

Goals are one way in which agents are programmed. Common goals include perform goals, achievement goals and maintenance goals. One limitation found currently with some agent systems is that their behaviour is predominantly reactive, especially with regards to maintenance goals. Many maintenance goals are reactive, which means they are only considered after a maintenance condition is violated. This leads to problems for issues such as safety or resource management, where these conditions should never be breached.

The approach developed in this thesis is to develop maintenance goals that exhibit proactive behaviour. This requires developing a representation of proactive maintenance goals, and identifying the mechanisms by which they can operate. The experimental evaluation conducted in this thesis shows that proactive maintenance goals consume less resources than reactive maintenance goals, and in error-free environments, can prevent maintenance conditions from being violated.

Agent Systems

One particular approach to developing intelligent agents is the notion of goal-oriented agents, in particular *Belief-Desire-Intention* (BDI) style agents. Agents are defined in terms of the goals that they aim to achieve, as designed by the agent developer or user.

There exists several frameworks that provide a foundation for developing intelligent agents. Some of these systems act as agent interpreters, whilst others compile agent specifications into executable code. Further to this, some frameworks build upon existing language constructs, while others introduce a completely new language targeted towards agent development. A more extensive introduction towards agent platforms can be found in Section 2.6.

There are many goal-types available in agent frameworks, as well as in theoretical approaches to agents. These include goals such as achievement goals, maintenance goals, perform goals, query goals and soft-goals. Each goal behaves in a different manner, but they all alter the manner in which an agent behaves.

The most common goal-type found in intelligent agents is the concept of achievement goals. In theory, these identify a particular state of the world that the agent would like to see come-about. In practice, achievement goals are commonly used as a mechanism for triggering a particular predefined routine, or *plan* that may bring-about the goal state upon completion. In some cases, these goals may best be described as *perform goals*, which are goals that cause actions to be performed, but do not check for a particular condition to be satisfied at completion.

Maintenance Goals

Another goal-type that is becoming more relevant is the maintenance goal. These define a particular state of the world that the agent should ensure is always present. In practice however, they are realised as triggers, so that a particular plan or action is activated when some particular condition (the *maintenance condition*) is no longer met. This behaviour is actually very similar to the behaviour of an achievement goal. The significant difference between achievement and maintenance goals is that an achievement goal is dropped upon successful completion, while a maintenance goal continues to be held by an agent, even when the particular maintenance state is re-established.

This is not the most rational behaviour however. Instead of waiting until a maintenance

condition no longer holds, an agent should determine if the maintenance condition will fail in the future, especially if this failure is caused by the actions that it performs. This approach of pre-empting failure can be described as a *proactive* approach towards maintenance goals. It is this concept of proactive maintenance goals that represents the driving force of this thesis.

For example, consider an agent that controls the temperature of a boiler. For safety, its aim is to keep the temperature of the boiler less than a certain point. If the temperature of the boiler is rising, it should act to reduce the temperature before its limits are exceeded. With reactive behaviour, the temperature would normally need to be exceeded before any response would be taken. With proactive behaviour however, once the agent detects that the temperature is likely to be exceeded, it would start performing actions then.

A similar example may apply to a mobile robot which consumes fuel as it moves around its environment. If it has a goal of moving to a distant location, it may need to refuel in order to do so. With reactive behaviour, it would only decide to refuel once it realises it is too far from a refueling location. With proactive behaviour however, before starting its journey to the remote location, it would first determine how much fuel would be consumed, and if it would need to refuel on this trip. If so, the agent would choose to refuel first, and then move to the distant location.

There is a need for proactive behaviour in intelligent agents, as they are slated as solutions for problems requiring autonomy. This is especially important when they are used in areas where resources are scarce or limited, as well as in situations where limits are imposed for safety or legal reasons. In this thesis, we will illustrate that maintenance goals with proactive behaviour are a suitable method for addressing these issues in goal-based agent systems.

1.1 Contributions

This thesis contains several contributions. Firstly, the required attributes of proactive maintenance goals will be determined. Secondly, a representation of maintenance goals is developed, and its use with a number of algorithms that illustrate how operation is performed will also been developed. Thirdly, operational semantics will be defined for maintenance goals (with both reactive and proactive behaviour) for the agent language CAN. Finally, the benefits of proactive behaviour, when compared with the existing reactive behavior in maintenance goals, will be evaluated. To achieve this, we use a scenario that utilises maintenance goals in the presence of reliable and unreliable data. This allows us to observe how these goals

behave, with reactive and proactive behaviours, and handle varying degrees of uncertainty, which is a core element of many agent domains.

In particular, the following research questions will be addressed:

- What properties are required to support proactive behaviour in maintenance goals? How can these properties be represented?
- How does an agent with proactive maintenance goals operate, in particular, in the presence of other goals, including maintenance goals? What are the operational semantics that capture this proactive behaviour?
- What benefit do proactive maintenance goals provide, compared to maintenance goals that are only reactive? How does the accuracy of its predictions influence the performance?

These questions will be answered using the following methodology. Case studies are developed that illustrate the current state-of-the-art with regards to maintenance goals. These are examined, and their limitations noted. We address these problems by utilising proactive maintenance goals, and examine their altered behaviour.

In order to do so, a definition of a maintenance goal with proactive behaviour will be developed. We examine existing notions of maintenance goals in order to develop a generic notion of what a maintenance goal is, and extend these to allow maintenance goals to exhibit proactive behaviour. This involves adding additional attributes to the maintenance goal construct, as well as new reasoning algorithms that were needed to provide proactive functionality.

Finally, we conduct experiments that compare and contrast maintenance goals with reactive and proactive behaviours. The experiments have involved the realisation of a case study, which has been explored using maintenance goals with reactive behaviour only, as well as maintenance goals with proactive behaviour as well. Variations in this case study have also been employed to identify cases where proactive behaviour is warranted, as well as cases where proactive behaviour may be less suitable. Our evaluation and findings in our experiment suggest that proactive maintenance goals consumed less resources than reactive maintenance goals in error-free environments, completely eliminating waster (avoiding

superfluous resource consumption). Specific types of errors (detailed in Chapter 6) can be avoided through the use of proactive maintenance goals. In environments with errors which cause an agent to overestimate, proactive maintenance goals may cause an agent to avoid starting goals it believes it cannot complete. While this is rational, it can be over cautious. In environments where underestimation occurs, reactive maintenance goals may cause impossible goals to be repeated ad infinitum, which is not rational behaviour. Further details concerning these experiments are detailed in Chapter 6.

This thesis is organised as follows:

Chapter 2 presents background material concerning intelligent agents, existing agent-platforms and their notion of goals. This includes an introduction to several styles of agent, including deliberative and reactive agents, as well as hybrid-style agents. As the focus in this thesis is on deliberative agents, a discussion of goals and plans is also presented. There is some discussion also on several popular agent frameworks, and how they represent and utilise goals. Additionally, we also discuss work similar to our own, reflecting on the similarities and differences between this and our own.

Chapter 3 begins by introducing a case study of an agent that utilises maintenance goals. The behaviours of such maintenance goals are analysed, and their limitations discussed. A preferred behaviour is determined, and the additional attributes that such an agent or maintenance goal would require is identified. This process is repeated for several case studies. The chapter continues by developing data structures that correspond with the improved representation of maintenance goals, supporting both reactive and proactive behaviours. When used in conjunction with algorithms also developed in this chapter, we show how the desired characteristics of maintenance goals are realised in such a system.

Chapter 4 contains formal semantics of the representation and algorithms developed in the previous chapter. By developing a formal specification of maintenance goals (with both reactive and proactive behaviours), proofs concerning several aspects of maintenance goals in an agent framework are developed. We provide additional background information on the agent language CAN, which is utilised in the construction of our contribution.

Chapter 5 consists of a sequence of experiments that aim to illustrate the benefits of maintenance goals with proactive behaviour. This involves introducing the experimental case study and the components that are varied through the course of the experiments. Each experiment consists of an introduction, explaining the aims, the results and finally a discussion of the findings. The chapter concludes with a summary, including possible ‘best-practices’ when utilising maintenance goals with reactive and proactive behaviours.

Finally, Chapter 6 provides some concluding remarks, and identifies several potential areas of future work.

Chapter 2

Background

This chapter reviews existing literature necessary for understanding the remainder of this thesis. It includes an introduction to agency and intelligent agents, including the Belief-Desire-Intention (BDI) paradigm, several goal types (in particular, achievement and maintenance goals), as well as goal interaction.

In Section 2.1, we provide an introduction to the field of intelligent agents. This includes a discussion on several key agent models which have been proposed. A discussion of the BDI model of agency follows in Section 2.2. Goals, in particular achievement and maintenance goals, and their properties, are discussed in Section 2.3, and an introduction to plans follows in Section 2.4. A discussion regarding goal interaction follows. This chapter concludes with an examination of maintenance goals in agent systems, both practical and theoretical.

2.1 Agents and Goals

Informally, a software agent is a computer program which performs tasks on behalf of a user or at the request of other agents. While this may describe many software applications, software agents share certain characteristics which differentiate them from non-agent software. However, there are several differing descriptions of the characteristics that identify an agent (for example, Maes [1994]; Wooldridge and Jennings [1995]; Franklin and Graesser [1997]; Georgeff et al. [1999]). The characteristics identified by Wooldridge and Jennings [1995] include many of the aspects shared by several descriptions.

Autonomy An agent operates to pursue its design objectives without human involvement.

Situated An agent is located in some environment which it can influence, and in turn,

be influenced. In many situations, an agent will exhibit only *partial* influence over the environment. This is apparent in situations involving multiple agents, each with their own design objectives.

Reactivity An agent responds, in a timely fashion, to changes in its environment.

Proactivity An agent can take the initiative, and exhibit goal-directed behaviour as required to achieve its design objectives.

Social Ability An agent has the ability to interact with users and other agents.

There are other characteristics that appear popular amongst agents, such as mobility and the ability to learn. These characteristics are not applicable in all situations however, hence they are not listed in the description above.

The ability for an agent to behave proactively, as well as reactively, is one of the most important features of agents. Reactive behaviour is well suited to situations requiring rapid (or near-instantaneous) responses, while behaviour that is proactive is best suited to situations that require long term strategies. Goals are essential for proactive behaviour in agents. Without goals, an agent has no long term aim; it simply reacts to the environment. For example, consider an agent in charge of the temperature of a boiler. It may have a goal of keeping or maintaining the temperature at a certain level *indefinitely*. To this end, it may act reactively at times when water is added or removed from the boiler, but the agent's overall purpose is to ensure that the temperature is at a certain level in the long run.

There are several types of agents. The type that we focus upon in this thesis are described as *deliberative* agents. Deliberative agents determine what it is they want to achieve by a process called *deliberation*. That is, deliberative agents form a goal towards some particular task, which they may frequently re-evaluate. While the work developed in this thesis may be applicable to many different types of agents, we focus on deliberative agents that utilise goals, as well as other mental attitudes such as beliefs and plans, as part of the reasoning processes.

2.1.1 Agent Architectures

There are several architectures presented for developing intelligent agent systems. These include deliberative, reactive and hybrid models. One deliberative model, the Belief–Desire–Intention (BDI) model of intelligent agents, as introduced by Rao and Georgeff [1995], is the

```

1  if both sensors detect obstacles {
2      move backwards
3  } else if the left sensor detects an obstacle {
4      steer to the right
5  } else if the right sensor detects an obstacle {
6      steer to the left
7  } else {
8      move forward
9  }

```

Figure 2.1: Example of a simple Subsumption Architecture based agent

focus of the work in this thesis. In this section, we will briefly discuss some alternate models, while in the following section, we will provide more detail concerning the BDI model.

Reactive Agents

The label ‘reactive agent’ is commonly used to describe an agent that does not employ a symbolic model of its environment. Instead, its behaviour emerges as a result of its simple behaviours that interact with the environment. This is in contrast to the deliberative agents which observe the environment, deliberate, and then act. In this section, we discuss several well-known models of reactive agency.

Rodney Brooks’ *subsumption architecture* is perhaps the best known reactive agent architecture. Brooks [1991b;c] suggested that intelligent behaviour can be generated without explicit representation or abstract reasoning, and that intelligence is an emergent property of certain complex systems.

In the subsumption architecture, an agent’s decision making behaviour emerges as a result of a set of task accomplishing behaviours. Each behaviour is intended to realise one particular objective, and maps perceptual input to actions, without any complex reasoning. Brooks [1991a] models this behaviour as a finite state machine.

One simple example of the subsumption architecture is to consider a small robot with several simple behaviours. The robot is equipped with two sensors, one that detects obstacles that are on the left side of the robot, and the other that detects obstacles to the right of the robot. Figure 2.1 lists possible pseudo-code for this simple agent. These simple behaviours produce an agent that is capable of navigating simple passage ways without colliding with walls. When an obstacle is detected, the robot steers away from it. In the event that it finds both sensors detecting an obstacle, the robot moves backwards and tries again, perhaps this time finding an alternate path.

At a similar time to Brooks, Agre and Chapman [1987] presented the PENGU approach to reactive agents. PENGU was a simulated computer game with a main character controlled via a reactive agent. In the PENGU architecture, many tasks were considered ‘routine’, in that they did not require much reasoning, and could be achieved in the same (or with only slight variation) method each time. Methods for achieving these tasks could be encoded in a very simple fashion, such as a digital circuit, which would only require updating to handle new situations.

Continuing the idea of representing agent behaviour as something as simple as a digital circuit, Rosenschein and Kaelbling [1986] introduce a compiler that converted a specification of an agent using declarative terms (such as goals), into a digital circuit.

In this way, some of the issues found in declarative agents, such as potentially slower response times, can be avoided. In this case, the notion of symbolic reasoning occurs at compile time, rather than run time. This approach is still considered reactive however, rather than proactive, as goal structures are not manipulated during execution, nor is any form of deliberation employed.

In a similar fashion to Brooks’ subsumption architecture, the agent network architecture presented by Maes [1989; 1991; 1990] uses a number of modules that exhibit simple, basic behaviour. An Agent Network Architecture approach, each module consists of pre- and post-conditions (in STRIPS-style of Fikes and Nilsson [1971]), and an activation level. The modules can then be linked based on their specified pre- and post-conditions. During execution, several modules may be active at any one time, and the activation levels associated with each module carries from one module to other modules they may be linked with.

Reactive agency has been shown to overcome some of the limitations associated with declarative agents, such as the difficulty in generating symbolic models of the environment. When compared with declarative agents, they are often computationally faster and can be simpler to produce, as the desired behaviour emerges from simpler behaviours.

The main features of these reactive agent architectures is that the agent interacts with the environment directly, eliminating the need for modelling the environment internally, and run-time planning is not required, as the agent’s behaviours are predetermined.

However, it is difficult to suggest that a reactive agent exhibits long term goal-directed behaviour. If the agent’s behaviour is determined solely from its reactions, the level of autonomy one would ascribe to such an agent would be quite low. From an agent development point-of-view, reactive agent architectures are potentially problematic. As the number of behaviours in an agent increases, there is a large increase in their potential interaction. This

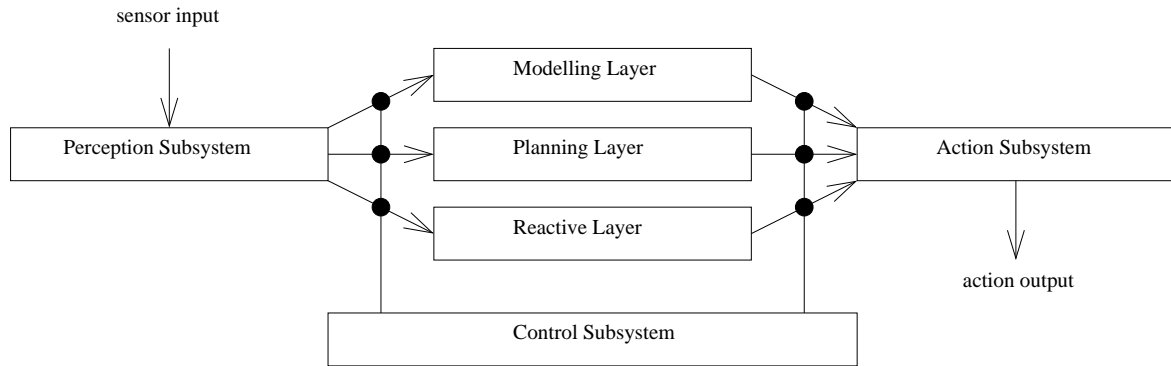


Figure 2.2: *TouringMachine Architecture*

can lead to difficulty in the understanding and diagnosis of problems.

Hybrid Agents

Hybrid agents combine the simple concepts found in reactive agents with the powerful reasoning capabilities found in deliberative agents. To incorporate these notions from both reactive and declarative agents, often by representing reactivity in one layer of the agent, and proactive behaviour in a separate layer, that are then linked in a particular manner, depending on the approach employed. Two such examples of hybrid agents are the TouringMachine and InterRap architectures.

The TouringMachine architecture by Ferguson [1992a;b] consists of several horizontal layers that are *activity producing*, in that each layer continually suggests actions the agent should perform. These layers, as illustrated in Figure 2.2, are denoted as the *reactive layer*, *planning layer* and *modelling layer*. An additional control subsystem selects which actions that are suggested the agent will ultimately pursue, as well as inhibiting sensory information for certain levels.

The reactive layer is used to provide an immediate response to certain situations as they occur, and is often represented as a mapping between percepts and action, in a similar fashion to the subsumption architecture’s task-accomplishing behaviours.

The planning layer is suitable for behaviour that may require more deliberation than simple reactive response situations. This layer does not plan in the traditional sense, but rather it employs a plan library of partial plans referred to as *schemas*. This layer is similar to the plan libraries often found in BDI agent architectures.

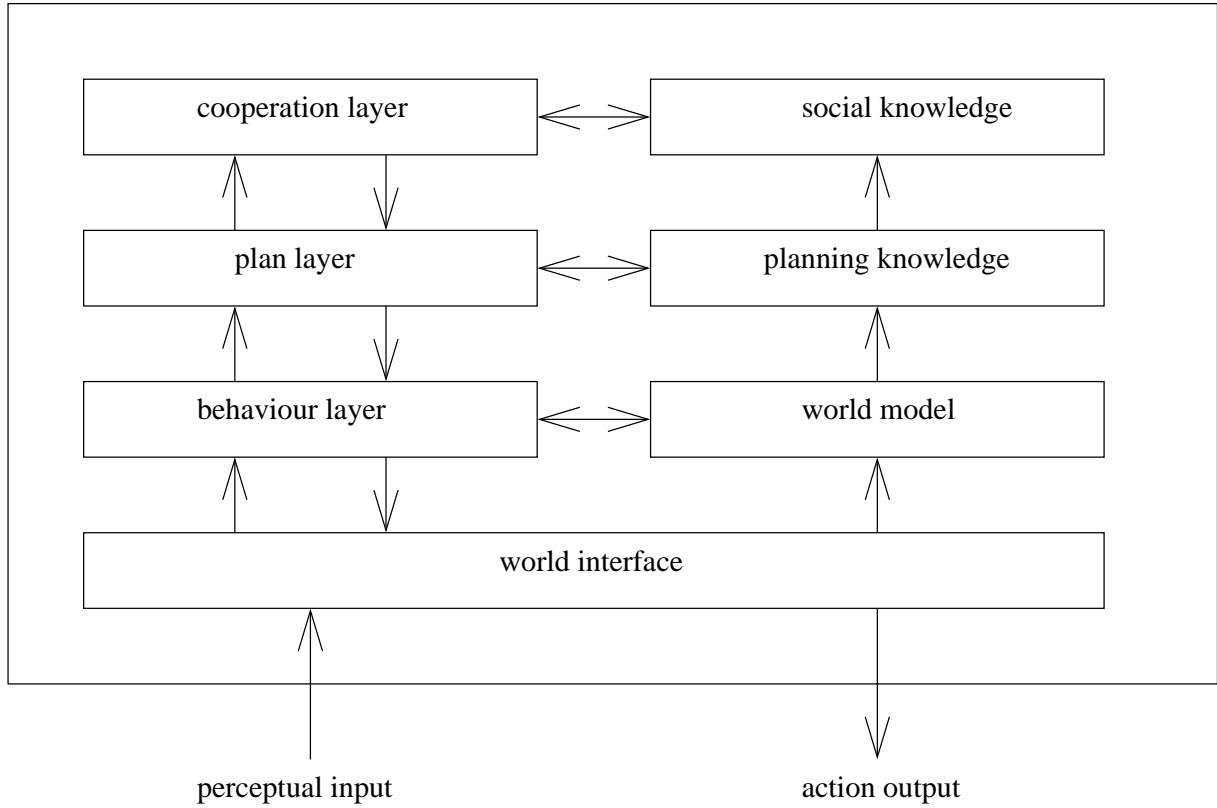


Figure 2.3: *InterRRaP Architecture*

The modelling layer produces a model of the environment that includes the agent itself, other agents and other entities in the environment. This layer is responsible for avoiding conflict and the generation of new goals for the agent to pursue. These generated goals are directed to the planning layer where the agent utilises its schemas to perform appropriate actions.

The InteRRaP architecture by Muller [1994]; Muller et al. [1994]; Muller and Pischel [1994], is a vertically layered hybrid agent architecture, and consists of several layers. As in the case of the TouringMachine architecture, InteRRaP layers also correspond to reactive, planning and cooperation (interaction with other agents). In contrast to TouringMachines however, where layers rarely communicate with one another, InteRRaP layers communicate with each other regularly. In this way, higher levels can utilise the beliefs, knowledge and goals of lower layers in their reasoning. In addition, each layer in InteRRaP also includes a knowledge base specific to that layer. Figure 2.3 illustrates the InteRRaP architecture.

During the execution of an InteRRaP agent, perceptual information is passed into the lowest layer, in this case, the reactive layer. If this layer is unable to deal with this percept, control is passed to a higher level, in this case, the plan layer. This behaviour continues until a layer is found that is suitable for this percept. Control is then passed back down and results in a particular action being selected and then executed.

Hybrid architectures aim to find a balance between reactive and deliberative agent architectures, aiming to obtain rapid responsiveness found in reactive systems, as well as the long-term goal-oriented behaviour found in deliberate systems. Layering provides an obvious method by which some of the desirable agent characteristics can be obtained, by allocating a layer to attributes such as *reactivity*, *proactivity* and *social ability*.

The layering approach is not without problems however. Overall behaviour can be hard to determine, due to the large number of behaviours that can interact. While horizontal layering approaches, as found in the TouringMachine architecture, are often conceptually easy to synthesize (with each layer corresponding to one particular behaviour that can be exhibited), the overall behaviour of the agent can be difficult to determine. In vertical layered architectures, such as InteRRaP, this problem is partially alleviated, as each layer interacts with a small number of other layers. However, interaction between layers still occurs, and the potential for complexity arising from this interaction is present.

2.2 BDI

One popular agent paradigm is the Belief-Desire-Intention (BDI) model. While the work developed in this thesis may be applicable to other agent models, it is most suited to BDI agents.

BDI has its origins in the philosophical work of Michael Bratman. Bratman [1987] focused on the role *intention* played in the reasoning processes found in humans. Rao and Georgeff [1991; 1995] formalised Bratman’s work, providing a strong foundation for agent theories and systems found today.

BDI agents are composed of the mentalistic concepts of beliefs, desires and intentions. *Beliefs* represent the information an agent has about itself and the environment (potentially including information about other agents). For example, a soccer playing robot may have the belief that it is located 10m from the ball, that the ball is located 5m from the goalkeeper, and that the goal-keeper has seen the ball. From these beliefs, it may determine that it cannot get to the ball before the goalkeeper does.

It is important to distinguish beliefs from knowledge. Beliefs represent information that the agent has determined via sensors or other inputs, and can related to the agent itself, other beliefs and goals of the agent, as well as beliefs about other agents in the environment. Relying on these sensors means that beliefs could in fact, be incorrect. Knowledge, on the other hand, is information known to be correct. For example, the information that the robot's id is `attacker1` is knowledge, where as the information that it is 10m from the ball is a belief.

Desires represent states the agent would like to have brought about. For example, the soccer playing robot desires the ball to be in the opponent's goal.

Another concept, similar to desires, are *goals*. Goals represent states of the world an agent would like to see brought about. A key difference is that the desires of an agent may be inconsistent with on another, where as goals are required to be consistent, as well as being possible to be realised. This is discussed by Rao and Georgeff [1995], as well as Bell and Huang [1997]. Goals will be further discussed in Section 2.3.

An *intention* acts as a commitment to realising a particular desire. Without intentions, an agent has no reason for pursuing an action. It is *irrational* for an agent to perform action without a reason (intention) for doing so. In a similar fashion, it is irrational for an agent to have an intention, but not perform action towards realising this intention. For rational behaviour, when an intention is formed towards some particular desire, an agent will act in a manner that will eventually bring about that desire. Intentions also differ from desires in that they should be consistent with one another – there is no point in the soccer playing robot intending both to attack and defend at the same time. It must choose one role at a time.

Many practical agent systems may represent intentions as being goals with *plans*. A plan represents some method of achieving a goal, and may be represented in many forms. When an agent selects a goal for pursuit, a plan will be instantiated which, when executed, should lead to the satisfaction of the goal. Plans will be discussed in greater detail in Section 2.4.

When an agent forms an intention, it does so with the belief that it will drop it when completed. However, an agent may have several intentions at once. These intentions may be inconsistent, or changes to the environment may mean that an intention is no longer possible. Therefore, an agent's selected intentions are subject to alteration. Rao and Georgeff [1995] discuss several commitment strategies, indicating when an agent may no longer hold an intention towards a desire.

- A *blindly* committed agent will not drop its goals until it believes them to have been satisfied. This means that if circumstances change (such as the environment) that makes it impossible for some goal to be satisfied, a *blindly* committed agent will not drop this goal – it will continue to make attempts at satisfying it.
- A *single-minded* agent will not drop its goals until it believes them to have been satisfied, or it believes that it is no longer possible to be satisfied. In either of these cases, the goal will be dropped. This is weaker than a *blindly* committed agent, but does not allow an agent to reconsider its goals.
- A *open-minded* agent maintains its goals until satisfied (at which time they are dropped), or until such time it decides to drop them. This may be in situations where goals interfere with other (possibly more important) goals, or that the agent determined that some goal is impossible. This allows an agent flexibility in choosing which goals it will pursue, and having the ability to ‘change its mind’ when appropriate.

An agent with *blind-commitment* maintains its commitment towards its intentions until it believes them to be true. Therefore, if a change occurs in the environment that makes some intention impossible, a blindly-committed agent will continue to hold its commitment towards this intention. This can lead to potentially stupid behaviour, as the agent will continue in its efforts to pursue a impossible goal.

A *single-minded commitment* agent holds its intentions until they are satisfied, or the agent believes it no longer possible. This change from blind-commitment allows an agent to reconsider its intentions, and drop those it cannot satisfy.

One possible problem with both blind and single-minded commitment is that they do not allow an agent to ‘change-its-mind’. Once an intention is formed, it must be kept until achieved (or until it is impossible in the case of single-minded commitment). A new intention that conflicts with existing intentions cannot be made, which may lead to inefficient performance. *Open-minded commitment* allows an agent to reconsider its intentions at any time, allowing an agent to drop intentions it no longer requires.

An agent may also have cause to reconsider its commitments. Experiments performed by Kinny [1990] used the terms *bold* and *cautious* to describe how often an agent may pause to reconsider its goals. Agents that rarely reconsidered were described as bolder, while those that did reconsider often, more cautious.

Kinny [1990] determined that in environments which changed slowly, bold agents per-

```

initialise-state();
repeat
    1.      options := option-generator(event-queue, B, G, I);
    2.      selected-options := deliberate(options, B, G, I);
    3.      update-intentions(selected-options, I);
    4.      execute(I);
    5.      get-new-external-events();
    6.      drop-successful-attitudes(B, G, I);
    7.      drop-impossible-attitudes(B, G, I);
end repeat;

```

Figure 2.4: Abstract Agent Interpreter

formed better when compared to cautious agents. This is due to the time that a cautious agent wastes reconsidering its commitments that remain the same. However, in environments that change quickly, cautious agents out performed bold agents, as they are able to determine earlier which intentions have no chance of success.

Rao and Georgeff [1995] also defined the operation of a system that exhibits BDI-like behaviour. The *abstract agent interpreter* (AAI) has been the basis of several practical agent systems such as PRS, developed by Georgeff and Ingrand [1989]; Lee et al. [1994] and dMARS, as discussed by D’Inverno et al. [2004].

Data structures representing the agent’s beliefs, goals and intentions are stored in B , G and I respectively. The operation of the AAI processes events, which are either externally generated by the environment, or internally generated by the agent. From this, a number of possible options may be generated (*Step 1*). The options are all possible courses of action.

These are similar to what we have earlier described as the desires, or the goals that are possible. These options are then filtered (*Step 2*), resulting in a set of selected-options. These are the things the agent has selected to realise, and are combined with the existing intentions in the I data structure (*Step 3*). The intentions in I are then executed (*Step 4*). Many practical agent systems represent these intentions as plans, but they make take any form. Once executed, new events that have occurred are placed in the event-queue (*Step 5*). Intentions that have been realised are then dropped (*Step 6*), followed by the intentions that the agent believes impossible (*Step 7*).

At the conclusion of the cycle, the agent will have executed some portion of its intentions. The new events that have occurred in the meantime, possibly as a result of the intention execution, will have been integrated with the agent’s belief base, and intentions that are no longer possible will be dropped. Additionally, intentions that have been completed are also

dropped. The cycle continues with the new beliefs being passed to the option generator in order to generate a new set of options.

2.3 Goals

Goals are important to many types of agents. In deliberative agents, there are many types of goals that are encountered. *Achievement goals* cause an agent to perform actions that lead the environment to some particular *state* of the world. Another type of goal is the *maintenance goal*. An agent with a maintenance goal aims to ensure some particular aspect of the agent or the environment continues to be satisfied.

The work in this thesis focuses on maintenance goals. We will examine the behaviour of maintenance goals currently found in intelligent agents, and identify ways in which they may be improved.

In agent systems, goals drive agent behaviour. For simplicity purposes in this thesis, we describe this as the goal having a certain behaviour. As an example, a maintenance goal may *behave* proactively. By this expression, we are suggesting that the maintenance goal will cause the agent to behave proactively.

2.3.1 Achievement Goals

Achievement goals are one of the most popular forms of goal found in agent systems. The purpose of an achievement goal is to bring about a state desired by the agent.

Some agent architectures treat goals as *goals-to-do*, as described by Winikoff et al. [2002]. In these frameworks, goals are used as triggers to plans. If a plan succeeds completely, the goal is also considered to have been achieved. This can lead to problems in some cases, where the plan succeeds but for some reason, the goal state is not reached. Using *goals-to-do* also makes it difficult to reason about concepts such as goal interaction, or if goals have been achieved by alternate means. Often in these frameworks, goals are only implicitly represented.

In contrast to this approach, some agent theories have proposed treating goals as *goals-to-be*. Such goals often represent the state of the world the agent is attempting to bring about. Such a mechanism allows the agent to perform reasoning over its goals, and determine if goals are already achieved.

In this thesis, achievement goals are treated as *goals-to-be*. Plans are associated with goals, which leads to actions being performed to realise the goal state associated with each goal.

The execution of the plan is dependent of the status of the goal state – the plan is stopped once the goal state is realised, and conversely, plan selection and execution is repeated if a plan is completed yet the goal state is not realised.

2.3.2 Maintenance Goals

When an agent achieves a goal, it is aiming to reach a particular state in the world where some condition is satisfied. However, an agent may also aim to keep a particular state true. For example, a soccer robot may aim to keep the ball away from the opponent. This is an example of a *maintenance goal*. In contrast to an achievement goal, where an agent aims to realise a particular state once, and then drop its commitment towards that goal, a maintenance goal aims to keep some condition true while the goal is adopted. In the event that the condition is no longer true, an agent with a maintenance goal acts to restore that condition.

A maintenance goal is appropriate in situations such as safety, or where repeated action may be necessary. An example from Pokahr et al. [2005b] describes a mobile robot that uses a maintenance goal to ensure that its battery’s charge is always greater than 10%. When this is no longer the case, the maintenance goal activates and causes the robot to find the closest recharger and recharge.

Unlike an achievement goal, a maintenance goal is *long-lived*, in that it will not be dropped upon success. Success for a maintenance goal is the continued satisfaction of the condition it is maintaining.

There are a variety of ways that this may be achieved, but they can all be described as *maintaining* a particular state.

2.3.3 Guarded Actions

One simple approach to maintaining a state is through the use of *guarded actions*, as found in agent system such as PRS, by Georgeff and Ingrand [1989]. A guarded action is often employed in conjunction with another goal, for example, an achievement goal. The purpose of a guarded action is to stop the achievement goal in the event that some condition (the guard condition) occurs.

If an agent aims to maintain a state, then this state can be used as the guard. While performing other actions in pursuit of the achievement goal, this guarded state should persist. In the event that it does not, the achievement goal is aborted. For example, the expression

($fuel > 10, moveTo(Location10)$) indicates that an agent should attempt to achieve the goal $moveTo(Location10)$ while the guard condition, $fuel > 10$ is satisfied. In the event that the guard condition no longer holds, the agent abandons the $moveTo(Location10)$ goal is aborted.

In some systems, in the event the guard condition fails, clean-up code can then be executed as a means to restoring the condition to maintain. Some practical agent systems such as JACK support maintenance goals in this manner.

In this manner, guarded actions exhibit reactive behaviour only. Action continues until the guard is no longer satisfied, at which point the overall goal is halted. Adding proactive behaviour with guarded actions would be difficult to address, as there is no link between the actions being performed and the guard (there is a link in the opposite direction however, as the guard can inhibit or halt the actions being performed). Therefore, alternate structures may be necessary for more complex reasoning and behaviours.

2.3.4 Reactive Maintenance Goals

An alternate to guarded actions are *reactive maintenance goals*, which are used in systems such as Jadex, by Pokahr et al. [2003]. Reactive maintenance goals, like many other goals, are adopted, and persist until some failure or drop condition occur. Unlike other goals such as achievement goals, reactive maintenance goals do not cause an agent to perform actions, as they remain passive until the maintenance condition is no longer satisfied; only at this point will they cause the agent to perform any action. The action performed generally has the aim of recovering or restoring the maintenance condition. If the maintenance condition is never violated, then this reactive maintenance goal will not influence the behaviour of the agent.

In some ways, reactive maintenance goals are very similar to achievement goals. Both types of goals are triggered and cause actions to be performed. The important difference between these two forms of goal is that the maintenance goal is not dropped upon success, like the achievement goal is.

Some practical agent systems such as Jadex support reactive maintenance goals. When some maintenance condition is met, action is triggered. Making this goal active may lead to the suspension of other goals, which can support behaviour similar to that found with guarded actions. Upon successful satisfaction of the maintenance goal, the suspended goals may be resumed. For example, monitoring the battery level in a mobile robot could be

realised with a maintenance goal. A reactive maintenance goal to perform such a task would involve having a condition that triggers the maintenance goal if the battery level was less than 10% (for example). Upon activation, the maintenance goal would cause the robot to stop what it was doing and recharge at the nearest base-station. However, this behaviour is only triggered when a certain value is reached – it does not take into consideration the current actions or goals of the agent. This can lead the agent to perform inefficiently, or potentially even becoming stranded.

Consider if the agent was located 1m from a base station and its fuel was around 15%. It would not act to recharge at this point. If it was given the task of travelling some long distance, it would select to achieve that goal. After travelling some distance, it may trigger the maintenance goal, causing it to need to return to the base station. This is an example of inefficient behaviour associated with reactive maintenance goals.

We see that maintenance goals with reactive behaviour extend guarded actions. Guarded actions do not introduce any new actions into the agent’s behaviour, and only halt existing actions. Maintenance goals with reactive behaviour *may* cause an agent to perform actions when necessary, for example, to refuel or resupply some resources when it runs below a certain point. Reactive maintenance goals also exhibit similar behaviour to guarded actions in that they both may cause other actions to be temporarily halted.

2.3.5 Proactive Maintenance Goals

As demonstrated in the previous section, there is a limitation inherent in treating maintenance goals only reactively; an agent will attempt actions that are destined to violate its maintenance goals, leading to inefficiency. One possible improvement to maintenance goals would be to have maintenance goals behave proactively – rather than waiting for the condition to no longer be satisfied, and *then* reacting to restore it, it may be better to act *before* the condition becomes unsatisfied.

There has been little progress in making maintenance goals behave in a proactive manner. In this section, we will discuss several methods by which this proactive behaviour has been exhibited.

Hindriks and van Riemsdijk [2007] was one approach to proactive maintenance goals, which utilised a method similar to a planning mechanism. If an agent determined that its currently selected course of action would cause a violation of one or more of its maintenance goals, it would abort that course of action. Thus, it proactively avoids its maintenance

goals from being violated. This does not provide any preventative mechanism however, other than simply not pursuing goals that cause the violation of maintenance conditions. One improvement to this would allow an agent to introduce actions that would allow it to achieve its goals, while ensuring its maintenance conditions remain satisfied (for example, forcing the agent to refuel its tank to allow it to complete a journey without violating a maintenance condition). Introducing this feature into this framework appears possible, but is not considered in that work.

Kaminka et al. [2007] suggested a multi-agent approach to proactive maintenance goals. A team maintenance goal may be to ensure that the distance between two mobile robots never exceeded a certain amount. Proactively, the robots could determine where other robots were heading and thus determine if the distance would exceed its maintenance conditions. If so, the robots would alter their plans to avoid this from occurring. The work presented in this thesis focuses on maintenance goals for a single agent. However, we believe that our findings are applicable to many multi-agent domains.

van Riemsdijk et al. [2008] is recent work concerning the representation and behaviour of goals in agent systems. Rather than providing descriptions of various goal types, this work presented a *generic* representation of goal, which was suitable for a large variety of goals. However, it is noted that the representation presented is insufficient to enable maintenance goals to be treated in a proactive manner. Reactive maintenance goals are possible within this framework, but exhibits the same flaws as discussed earlier in this chapter.

The work in this thesis will build upon the framework by van Riemsdijk et al. [2008] to provide maintenance goals that exhibit reactive and proactive behaviour, that is, maintenance goals that aim to prevent, as well as recover from, maintenance goal violation. A more extensive discussion of van Riemsdijk et al. [2008]’s work will be discussed in a later chapter.

Having introduced maintenance goals as found in several existing agent systems, it is clear that there is a potential for maintenance goals with proactive behaviour. Before examining this in more detail however, we will first explain some additional technical background regarding maintenance goals. This will be achieved over the next sections.

2.3.6 Alternate Approaches

The KAOS goal-oriented requirements engineering framework by Darimont et al. [1997], includes maintenance goals, as well as avoid goals. These goals are related, being the converse of one another. Where as a maintenance goal aims to keep some condition satisfied, the

purpose of an avoid goal is to *not* let some condition become satisfied. Modelling an avoid goal with a maintenance goal (by negating the condition to avoid/maintain), it becomes even more apparent how important it is to take the proactive approach concerning maintenance goals. If an agent wishes to avoid a condition from occurring, it will more than likely need to act in advance (behave proactively) rather than reactively. This has significance in situations such as avoiding breaking some legal requirement, or to ensure safety.

The work of Nakamura et al. [2000]; Baral and Eiter [2004]; Baral et al. [2008] focuses on defining exactly the behaviour of a maintenance goal, utilising temporal operators. They identify that the temporal operators, *always f*, is too strong. This does not describe the behaviour of a maintenance goal, where it is expected that maintenance conditions may fail, which then need to be repaired. Further, in many cases, it is impossible for an agent to exhibit such a high degree of control over the environment to guarantee *always f*.

One alternate proposed by Nakamura et al., is that of *always eventually f*. This encodes that if *f* becomes false at any point, it will eventually be (re)satisfied. This encoding is also dismissed by Nakamura et al. as being too strong. It is possible that an agent may be overwhelmed with requests such that it cannot restore the condition it is aiming to maintain. An example presented by Nakamura et al. is that of an agent that monitors a user's inbox, with the maintenance goal of keeping it empty. The environment is adversarial, keeping the inbox full and thus continually falsifying the maintenance goal. The agent removes and processes each email at a slower rate than the environment sends email. Therefore, despite the best efforts of the agent, it is unable to maintain the condition of keeping the inbox full. Although not satisfying the condition, this behaviour would still be described as rational, and attempting to maintain its goals, and is arguably the correct behaviour to be employing in this case, exhibiting perpetual action in attempting to satisfy its maintenance conditions.

Nakamura et al. [2000]; Baral and Eiter [2004]; Baral et al. [2008], define the notion of *k-maintainability* with respect to maintenance goals. *k-maintainability* describes the window of opportunity required in order for an agent to perform a number of actions that would *restore* the maintenance condition. A period less than *k* does not guarantee that an agent can maintain the desired condition, while given at least *k* steps occasionally, will result in the agent attempting to restore the maintenance condition. Nakamura et al. [2000]; Baral and Eiter [2004]; Baral et al. [2008], go on to represent and solve the problem of determining *k-maintainable* controls using a SAT encoding, which is then proven to be polynomial time, and linear time for small *k*.

A discussion of maintenance goals found in practical agent systems will be presented in

Section 2.6.

2.3.7 Other Goal Types

There are many other types of goals encountered in the agent literature. These include goals such as query goals, which grant agents the ability to update their belief base with additional information. The work presented in this thesis does not involve these types of goals directly, hence the discussion of these goals is limited.

2.3.8 Properties of Goals

We have discussed several types of goals that are commonly found in agent systems. In this section, we will focus on some common properties that are identified in agent literature concerning goals. These goal properties include that goals are *consistent*, *persistent*, *known*, *possible*, *unachieved* and *permissible*. Each of these properties will now be discussed.

Consistent

An agent may have several goals at any one time. It is important, therefore, that all these goals can be jointly satisfiable. We say that the agent's goals are *consistent*. To act rationally, an agent should not concurrently pursue goals that hinder each other's progress. Instead, it should postpone some of the goals until they no longer cause such interference.

In the work of Rao and Georgeff [1995], goals are expected to be consistent. However, mechanisms for ensuring that goals are consistent are not present.

Bell and Huang [1997] require that goals are *coherent*, in that they are jointly realisable with the goals that the agent considers more important. This requirement is reflected in their logical framework.

It is important that this goal be attempted when it is possible to do so. Such behaviour relates to goal *persistence*.

Persistent

When an agent adopts a goal, it should not be dropped (or no longer pursued) without good reason. Such reasons include the goal becoming satisfied, or changes in the environment make it impossible for a goal to be achieved. This behaviour is described as *goal persistence*, as an agent continues to satisfy a goal until it has good reason to stop.

Bell and Huang [1997] approach this by utilising a *goal hierarchy*. From all the goals available to an agent, some of these goals would be *preferred* by the agent. These preferences may be partial, and an agent will aim to maximise the goal hierarchy by selecting the goals

that are coherent. Goals may be dropped in favour of more preferred goals, if they are not coherent. Goals *persist*, in that the agent will continue to pursue them, until such a time they are no longer possible, have been achieved, or have been dropped in favour of more preferred goals.

As discussed earlier in this chapter, Rao and Georgeff [1995] discuss goal persistence in the form of identifying several commitment strategies, including *blind*, *single-minded* and *open-minded* commitment.

In all cases, goals are dropped once satisfied. Hence, for a goal to be adopted, it make sense that it is not already satisfied. This property may be described as a goal being *unachieved* prior to adoption.

Unachieved

An agent should not form a commitment towards a goal if the goal is already satisfied – the agent has nothing to do (with respect to this goal). Therefore, an agent should only adopt a goal if the goal is *unachieved*.

This property is particularly applicable to achievement goals. If a goal is already satisfied, it should not be adopted. For example, if an agent is located at a particular location, it is useless to adopt a goal of moving to this location. It is already satisfied.

Possible

Following from the property of *persistence*, goals must also be *possible*. For a goal to be adopted by an agent, it is rational that the agent have some means of achieving the goal.

In the work of Rao and Georgeff [1995], an agent may only adopt a goal if it believes it can achieve it.

van Linder et al. [1995] term this property *implementable*, and requires that there be a sequence of actions that an agent may take that would satisfy the goal.

Bell and Huang [1997] name this property *realisable*, which requires that an agent have a plan for achieving the goal. Furthermore, while a goal may be possible, it may not be *permissible*.

Permissible

Bell and Huang [1997] require goals to be *permissible*, that is, these goals do not violate any moral or legal constraints imposed upon the agent.

Known

Rao and Georgeff [1995] require that an agent *know* what its goals are. Having this information available to an agent may allow it to make more rational choices for which goals it intends to pursue, and at what times. For example, goals which interact in a positive way

(as discussed by Thangarajah et al. [2003b]) may be adopted at the same time, while goals that negatively interact (as discussed by Thangarajah et al. [2003a]) may require that they are adopted in a sequential manner. Such deliberation can only be realised when goals are known to an agent.

In some systems, an agent may not be aware of (know) its goals. This can lead to problems as an agent cannot reason over multiple goals to determine if they interact or conflict in some manner. More details concerning goal interaction will be addressed in following sections.

2.4 Plans

While goals describe *what* it is an agent is aiming to do, often it is the plans that describe *how* these goals will be realised. By following a particular plan for a goal, it is likely the goal will be realised. It is not guaranteed, however, as changes in the environment or unreliable actions, may cause steps in a plan to fail. In these cases, it is appropriate for the plan to be retried, or a new plan selected.

Plans may be generated on-the-fly (for *planning agents*), or possibly selected from a large number of plans available to an agent (*a plan library*). The latter is commonly found in BDI-style agents.

2.4.1 Goal-Plan Trees

In general, goals in BDI-style agents are accomplished by executing plans. Every goal has at least one plan, and a plan may also include sub-goals. This builds a tree structure with the top-level node representing the top-level goal, and plans broken down into branches. Thangarajah et al. [2002a] refer to this tree structure as a *goal-plan tree*.

The root of the tree represents the top-level goal. Branches extending from this node represent plans that can satisfy this goal. Children of these plans represent *sub-goals*. In plans, sub-goals may be combined in a variety of ways, including running in parallel or sequence. OR-branches can represent cases where only one branch is required to succeed, while in the case of AND-branches, all branches must be satisfied for this node to also succeed.

With goal-plan trees, actions are considered atomic and are not present in the tree. Instead, their effects, pre-conditions and in-conditions can be stored at the node level. Therefore, the goal-plan tree represents a tree showing possible paths by which a goal may be realised, through a variety of plan choices and paths. This information could later be used for more detailed reasoning, as we describe in the following section.

Figure 2.5 illustrates a goal-plan tree. The top level goal, `GoToWayPointGoal` can be realised by one of two plans, `DriveThruCanyonPlan` or `DriveAroundCanyonPlan`. Both plans bring about the effect of `AtWayPoint`. However, the first plan results in 10 units of fuel being consumed, while the second plan results in 20 units of fuel being consumed.

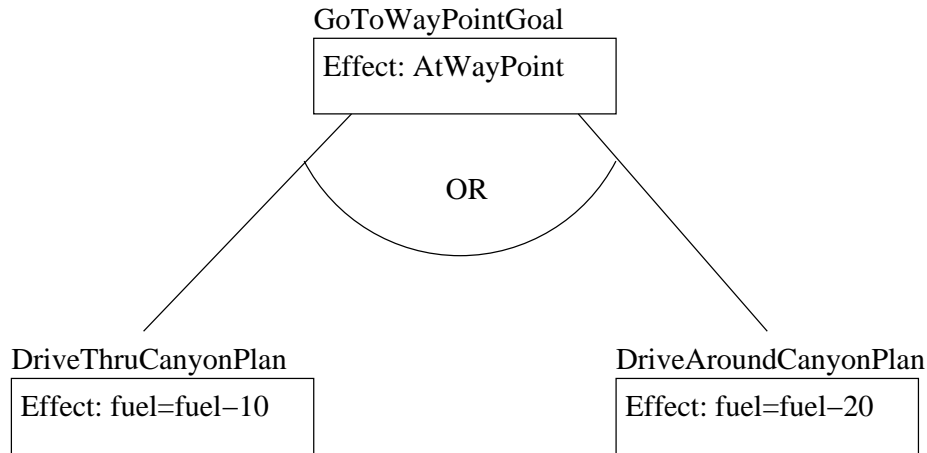


Figure 2.5: An example of a goal-plan tree

2.5 Goal Interaction

An agent has the ability to pursue multiple goals simultaneously. However, in order to do so, it must be possible that these goals can be executed at the same time. There are situations that arise that prevent multiple goals from being realised simultaneously. For example, an agent cannot be in two locations at once, nor can it kick a ball in several directions. Furthermore, there may be times when multiple goals interact in a positive manner. If several of an agent's goals require it to move to a particular location, this need be only performed once, if the agent can detect this and ensure that both goals are realised when this occurs. Thangarajah et al. [2002a] describe this feature as *goal interaction*.

One way of performing goal interaction (and the method prescribed by Thangarajah et al. [2002a]) is to expand upon the concept of goal-plan trees. Utilising the hierarchical information found in these trees, information pertaining to the sub-goals and plans may be propagated up the tree to a higher-level goal. This information is described as *summary information*.

Thangarajah et al. [2003a;b; 2002b] describe several ways in which goals may interact.

Goals interact positively or negatively, and interact over the resources consumed and produced, and the effects generated by the goals.

2.5.1 Interaction due to resources

In many agent scenarios and applications, the resources available to an agent are finite. Therefore, there may be times when an agent has only limited resource available, while having several parallel goals to achieve. Only some of these goals may be achievable with the resources available.

Resources may be categorised in several ways. Thangarajah et al. [2002b] regard resources as *consumable* and *reusable*. Consumable resources are eliminated once used by an agent, while a reusable resource may be reused for the purposes of another goal, just not simultaneously.

From this, algorithms by Thangarajah et al. [2002b] allow for resources to be considered as either *necessary* or *possible* resources for achieving a goal. Necessary resources are resources that are used in all ways of achieving a goal, while possible resources are only utilised in some of these ways.

The resource requirements for a goal can be derived by combining the resource requirements of all possible plans that achieve that goal, while the requirements of a plan can be determined by combining the resource requirements of that plan's sub-goals, and the actions performed by that plan. Using the goal-plan tree, the resource-requirements for any high-level goal can be determined in this manner.

Utilising this resource summary information, there are three cases that can be determined regarding if goals are in conflict over their resources. The first is that the goals are in *conflict* if it is impossible to achieve all goals with the currently available resources. The second case is that the goals are *safe* if all goals can be satisfied, regardless of the order in which the goals are executed. Finally, all goals *may* be achievable given a particular ordering of goals (*schedulable-dependent*), or if the algorithms are unable to determine the outcome, these goals may be considered *uncertain*.

2.5.2 Interaction due to effects

When an agent developer writes a plan, often they will use earlier plan steps to establish conditions for later plan steps. The effect of these steps is referred to as *preparatory effects* by Thangarajah et al. [2002b]. Plans may also require *in-conditions*, or conditions to remain

established during the execution of a plan.

Goals may interact over their effects due to these cases. As in the case for resources, algorithms allow an agent to determine the *definite-conditions* and *potential conditions* that occur in the execution of a goal. Similar mechanisms for identifying cases where interference occurs can also be established.

Interaction due to effects can also work positively. If the effects of some sub-goals for different goals in an agent are the same, then there is no need for all these sub-goals to be executed. Instead, if an agent is able to schedule the goals to be executed at the correct times, in some cases, only one occurrence of the sub-goal would be required.

Thangarajah et al. [2002b] have developed mechanisms and algorithms for determining if goals interact. However, this work is limited to achievement goals only. The work in this thesis aims to address the way in which maintenance goals interact with other goals in an agent.

2.6 Maintenance goals in practical agent systems

In Section 2.3.2, we discussed maintenance goals. In this section, we discuss some of the popular families of agent platforms, and how goals, in particular, maintenance goals, behave and are represented in these frameworks.

2.6.1 PRS, dMARS, JACK

Due to their shared heritage, the PRS family (PRS, dMARS and JACK) share many similar notions concerning goals. In this family, goals are not explicitly represented, rather, they are implicitly captured via events. When a particular event is received by the agent, plan selection occurs and a plan appropriate to this event is then pursued. In the default behaviour, if the plan being pursued fails, an alternate plan is selected and the process repeats. The ‘*BDI-gap*’ described by Winikoff et al. [2002], is partially a result of not having an explicit notion of goal.

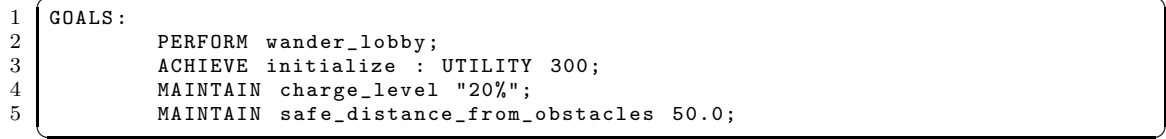
While achievement goals are the most common form of goals, the PRS family includes several alternate goal types including maintain and query goals. The behaviour of a maintain goal is to trigger an event (goal) when the maintain condition does not hold during the attempted achievement of the sub-goal. Upon completion of the sub-goal, the agent drops this maintenance goal. Hence, the maintain goal is dynamic, in that it may be adopted and dropped during runtime. If the maintain condition becomes unsatisfied during the execution

of the sub-goal, the sub-goal is aborted and recovery actions may be pursued. However, the original goal is dropped, and must be explicitly re-adopted if the agent wishes to continue pursuit of this goal.

2.6.2 JAM

The JAM Agent language by Huber [1999] builds upon the UMPRS (discussed by [Lee et al., 1994]) and PRS (by [Georgeff and Ingrand, 1989]) implementations of the PRS agent framework. It supports a variety of goal types, including Achieve, Perform and Maintain. However, the form of maintain goal represented in JAM is reactive in nature.

An example of a JAM agent described by Huber [1999] can be found in Figure 2.6. After executing the `initialize` action (which has the highest utility and is thus selected in preference to all other goals), the agent continually performs the `wander_lobby` action, ensuring that its charge level is always greater than 20% and keeping a safe distance from obstacles.



```

1 GOALS:
2   PERFORM wander_lobby;
3   ACHIEVE initialize : UTILITY 300;
4   MAINTAIN charge_level "20%";
5   MAINTAIN safe_distance_from_obstacles 50.0;

```

Figure 2.6: Example Goals in JAM

While wandering the lobby, an agent with this goal set will disregard the maintenance goal of ensuring that its charge level is greater than 20%. It is only when this condition no longer holds will this maintain goal come into effect and cause the activation of a plan that restores this condition. This is therefore, a reactive maintenance goal.

A similar example may also apply to the same lobby wandering agent in the presence of obstacles. To avoid collisions with obstacles, maintenance goals could be used, which are triggered when an obstacle is too close (less than 50.0 units) to the robot.

2.6.3 Jadex

Jadex is a Java based agent language. Originally built upon JADE¹, a FIPA² compliant framework for hosting and developing multi-agent systems, Jadex provides a framework for

¹More details concerning JADE can be found in work by Bellifemine et al. [1999]

²More details on the FIPA organisation appear in work by O'Brien and Nicol [1998]

developing BDI based agent systems. Jadex provides a comprehensive collection of goal types, including achieve, maintain, perform and query. Further details of Jadex's goal types can be found in a comprehensive discussion by Pokahr et al. [2003]. This discussion also provides considerable detail concerning the representation of goals in the Jadex language. Figure 2.7 provides a description of the behaviour of maintenance goals in Jadex.

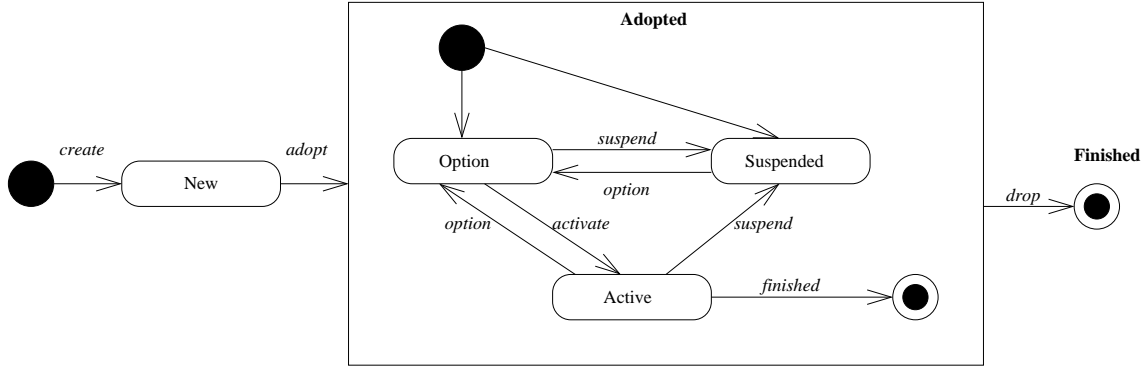


Figure 2.7: Maintenance Goal Life-cycle in Jadex (Taken from Pokahr et al. [2003])

Any goal in the Jadex framework is in any one of three states at any one time.

Option This corresponds to the case when this goal can be selected for pursuit, but has not yet been made active. This loosely corresponds to the concept of desire, in that it is a goal the agent would like to pursue, but is currently not taking action towards. This allows goals that are Options to be conflicting.

Active Once an agent decides to take action towards achieving a goal, the goal is then considered Active. This indicates that the agent is taking steps towards realising this goal.

Suspended A Suspended goal indicates a goal that had been Active, but for some reason, cannot be allowed to continue. It is therefore moved to the Suspended state. After a certain condition is met, indicating the goal is once again applicable, it is moved to the Option state where it may be selected when the agent deems it appropriate.

When a maintenance goal is adopted by an agent, it begins in the Option state. In general, it becomes Active when there is no other Active goal that conflicts with it.

Once Active, it monitors particular beliefs of the agent, triggering a plan when its condition is no longer satisfied. In the event that the agent deems that the condition is no longer

possible to maintain, the maintenance goal can be moved to **Suspended**, pending a possible change to the **Option** state when the maintenance goal is once again applicable.

Goals are explicitly represented in Jadex, which enables more complex deliberation when compared with some other agent families. However, there are no provisions for reasoning over maintenance goals other than simple inhibition links, as discussed by Pokahr et al. [2003]. Much like the behaviour found in JAM agents, the (default) behaviour of JADEX maintenance goals is to act as a trigger to action when the goal’s maintenance condition is no longer satisfied.

We now move to discussing some of the logic based agent frameworks.

2.6.4 AgentSpeak and Jason

One of the origins AgentSpeak(L), by Rao [1996], is to provide a mechanism for overcoming the ‘*BDI-gap*’, caused by the fact that practical implemented agent systems often diverged from the theoretical approach.

AgentSpeak(L) aims to formalise the operation of existing practical agent languages, that being PRS (and to some extent, its successor, dMARS) which had “*lacked a strong theoretical underpinning*” [Rao, 1996]. It achieves this by representing much of the BDI model in a first-order logical language, containing events and actions.

In Rao [1996], only two forms of goal are considered: *achieve-goals*, which are the common form of goal found in almost all agent platforms, and *test-goals*, which allows an agent to determine if it a particular formula is true or false relative to its belief set.

Achievement goals have a context related to them, which must be satisfied before the body can be executed. This context could be utilised to prevent actions from being executed – it does not support any method by which alternate actions could be performed if necessary.

Jason, developed by Bordini and Hübner [2005], is a Java-implementation of AgentSpeak(L), which supports *triggering events*, to indicate when a plan should be executed. Utilising this notion could provide support for a form of reactive maintenance goal in Jason.

Indeed, in Hübner et al. [2006], a schema for maintenance goals in AgentSpeak(L) is given, that acts as a reactive maintenance goal. Figure 2.8 illustrates this example.

These rules (and an associated plan that is not listed here) act to cause the agent to believe that its battery is not charged when the level is less than 20%. Dropping this belief can be used to trigger a goal or plan, which is used to direct the agent to recharge. Once the battery level reaches 100%, the second rule causes the adoption of the belief, **batterycharged**, which

```

1 +batterylevel(B): B < 0.2 --> -batterycharged.
2 +batterylevel(B): B = 1.0 --> +batterycharged.

```

Figure 2.8: Example Goals in Jason

can be used to stop the recharging plan.

This behaviour is completely reactive, and thus Jason (and AgentSpeak(L)) share the same limitation expressed in discussions concerning other agent systems.

2.6.5 3APL, GOAL and Dribble

The 3APL family of agent programming languages, developed by Hindriks et al. [1999]; Dastani et al. [2000], utilise constructs such as an agent’s beliefs and goals, in conjunction with a set of ‘practical reasoning rules’ which revise an agent’s goal set. 3APL also has facilities for *creating and modifying plans* during the execution of an agent.

An initial extension called GOAL had the planning ability found in 3APL removed, but can *use declarative goals* in selecting actions to perform. This was to address inability to determine if goals were completed by alternate means, and to enhance an agent’s ability to reason over its goals.

A later extension by van Riemsdijk et al. [2003] called Dribble, aims to consolidate the procedural and declarative aspects of the agent languages GOAL and 3APL. To this aim, it features the ability to plan with declarative goals, meaning that it can (in theory) perform more complex reasoning when compared with the original 3APL. However, as Dastani et al. [2003] reported, Dribble is limited in that it is a propositional language that excludes variables, limiting its practical use.

Incorporating the extensions developed in Dribble, 3APL was extended to include declarative goals and first order features.

Representing maintenance goals in 3APL can be represented by means of practical reasoning rules to activate certain goals when its maintenance conditions are violated. As discussed in Hindriks and van Riemsdijk [2007], maintenance goals can be represented in GOAL with some small modifications. In this paper, maintenance goals act mainly to *constrain* the actions available to an agent, and so, can be considered proactive. However, in order to achieve proactivity, the agent requires a look-ahead operator (that potentially needs *infinite* lookahead), in order to determine the consequences of its actions. This degrades the usefulness of

such an approach in terms of implementing proactive behaviour in practical agents, however, it is a useful approach for analysing such behaviour.

This later approach was then extended by Hindriks and van Riemsdijk [2008] to further extend support for distinguishing between *hard* and *soft* constraints, and preferences, that allow an agent developer to define which goals to pursue in favour of others. This rational action selection architecture (or *RASA*) realise hard constraints with maintenance goals, and soft constraints via preferences.

Summary

In this section, we have discussed several agent frameworks. We have seen that many utilise a notion of maintenance goal in both design and run-time.

However, many of the agent frameworks treat maintenance goals as triggers to the adoption of a goal or plan. In this way, they are similar to achievement goals, which are adopted in the case of a particular condition and which cause action to occur.

This chapter has presented several facets concerning maintenance goals in intelligent agent systems. As discussed, maintenance goals in existing agent literature have generally focused on reactive maintenance goals – maintenance goals which caused an agent to perform actions to restore a maintenance condition *after* the maintaining condition was violated. This is a severe limitation on the behaviour of an intelligent agent, and has significant ramifications for situations where limits should not be broken, for example, in areas including safety or performance.

The remainder of the thesis will aim to improve on the current state of the art concerning maintenance goals. The improvements we present will result in maintenance goals that more closely match the behaviour of *maintaining* a particular state. This will involve making an agent *proactively* perform actions that will prevent maintenance conditions from becoming violated.

Chapter 3

Representation and Reasoning

In this chapter, we introduce proactive maintenance goals via case studies, showing how current methods of maintenance goals are inefficient. We discuss the desired behaviour of each case study, and illustrate how proactive maintenance goals support this. This chapter concludes with a summary of the characteristics of proactive maintenance goals that we have identified.

3.1 Case Studies

In this thesis, we will discuss maintenance goals through the use of several case studies. These have been selected as they identify various behaviours of maintenance goals that we have deemed important.

There were several case studies we considered for demonstrating the features of maintenance goals, including an automatic intelligent refrigerator and a boiler. Ultimately, we focus our discussion on two; a Mars rover and a soccer playing robot. These case studies encompass many of the features found in the other case studies, as well as case studies found in agent literature.

3.1.1 Mars Rover

A Mars rover is a mobile robot capable of traversing a planet such as Mars. In this thesis, we will utilise a Mars rover to discuss how an autonomous rover may benefit from the use of proactive maintenance goals.

First, we will describe our version of the Mars rover. It is a wheeled robotic platform, with a fixed capacity fuel tank. As the rover moves about the environment, it consumes fuel.

For simplicity, we assume a linear relationship, so that one unit of fuel is for every one unit of distance travelled. For example, if the rover moves 3 units of distance, it consumes 3 units of fuel in the process. Additionally, no fuel is consumed for braking or turning. While this is not realistic, it aids in explanation and simplifies experiments, and could be expanded for future, more detailed work. To acquire more fuel, we assume that there is a refuelling depot on the planet, where the rover can go and refill its tank to maximum capacity.

Such a rover may interact with its base in the following manner. Rather than specify the exact actions to perform, as is done currently, engineers controlling the rover may specify at a high level the goals to be achieved. Some example goals may be to move to a specified location, or gather a sample of soil from another location. It is then up to the rover to perform the necessary actions to achieve these specified goals.

Maintenance Goals with the Mars Rover

As the rover accomplishes its goals, it will need to manage its fuel usage. If the rover was to run out of fuel, it would become stranded with no method of returning to the depot to gain more fuel. In addition, it should not spend all of its time at the refuelling station.

Maintenance goals may be employed to manage fuel in this scenario. The simplest form of maintenance goal to ensure that the rover does not run out of fuel, is to ensure that the fuel in its tank is always above a certain threshold, for example, 20% capacity. Let us represent this by the following *maintenance condition*; $fuel > 20$. While this condition is satisfied, the rover can continue performing other actions. However, if this condition ceases to be satisfied, the rover will act to fix it.

In order to fix this situation, the agent should pursue some course of action, which we identify as the *recovery goal* of the maintenance goal. In this particular case study, the *recovery goal* requires the agent move to the depot and refuel. When the *maintenance condition* is not satisfied, the rover may suspend all other goals and adopt the *recovery goal*. This goal is then executed. Once complete, it may resume the other goals.

There are certain problems with this approach however. The first is identifying what a suitable threshold value is. In this case, 20% has been selected at random, but is possibly inadequate. It is possible that the rover may be located in a position that requires more than 20% of the maximum fuel to return to the depot. It is likely that the rover could become stranded if it had just completed a long trip to a goal, and was too far from the depot. Clearly, to be safe, this triggering level should be set at 50%. That way, the rover

can guarantee that it is always a safe distance from the depot.

However, with a triggering level of 50%, it is likely that the rover will spend a lot of time travelling to the depot and refuelling. Clearly, this is inefficient.

This form of maintenance goal was presented in [Pokahr et al., 2003] as an example of using maintenance goals to manage refuelling for a (simulated) security robot.

One alternative to this approach is to use a more complex maintenance condition. We assume a linear direct relationship between fuel usage and distance, a maintenance condition could be $fuel > distance(depot)$, which is satisfied if there is enough fuel to move to the depot. The behaviour is different to the previous maintenance goal. Instead of relying on the 20% to trigger when to return to the depot, the rover can determine how much fuel is required to return to the base, and only refuel when it must do so (in practice, this level should include tolerance levels for safety).

In these examples, we see that the behaviour of the agent is reactive with respect to its maintenance goals. The rover can now move about its environment, and only when it is necessary, will it return to the depot to refuel.

This reactive approach has limitations. The most important is that the maintenance goal only comes into effect *after* the maintenance condition has been triggered. An alternate, possibly more efficient solution could be to prevent the maintenance condition from being violated in the first place, by means of preventing action from occurring.

For example, consider the rover located at the depot and needing to move 20 units away. It currently has 20 units of fuel. If the rover moves towards its goal, once it has moved 10 units, the maintenance goal will be triggered. It will return to the depot, refuel, and then resume moving to its location.

A more efficient solution to this problem would have the agent recognise that attempting to move to the goal 20 units away would cause the maintenance goal to be violated in the current circumstance, so to perform an alternate action instead. In the following section, we will discuss and expand upon this approach.

Maintenance Goals with Proactive behaviour

The approach we promote in this thesis is to make maintenance goals *proactive*. The maintenance goals we have presented so far have been *reactive* in nature. These maintenance goals have some condition which when no longer held cause the agent to alter its behaviour, often to restore the condition. While the condition is satisfied, the maintenance goal has no

influence over the behaviour of the agent.

While maintenance goals with proactive behaviour are active, they influence the behaviour of an agent. The agent will determine if it can pursue new goals with the consideration to its maintenance goals. Prior to adopting a new goal to visit some location, the agent determines if its maintenance goals will still be valid upon the success of the goal.

In the Mars rover scenario, a proactive-behaviour maintenance goal to maintain enough fuel to return to depot appears similar to the second maintenance goal described in the previous section, $fuel > distance(depot)$. However, the behaviour is quite different. Before the agent begins moving to a new location, it determines how much fuel would remain when it arrives at the goal. If this is not enough to return to the depot, there is no point in attempting to move to the location – the reactive maintenance will be triggered en route, and hence it will need to refuel. It is better to refuel initially, and then pursue the goal. Here, this goal is referred to as the *preventative goal*, as its purpose is to prevent failure, rather than recover from it.

For example, consider a Mars Rover with 20 units of fuel in its fuel tank, that has a capacity of 100 units. It is currently located 10 units away from the refuelling depot. The rover is about to adopt a goal to move to location A, which is 10 units away from its current location, C, and 20 units away from the refuelling depot, D (refer to Figure 3.1).

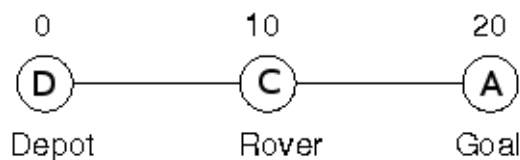


Figure 3.1: Scenario Overview

We first consider how a rover would handle this scenario when using maintenance goals with reactive behaviour. The maintenance goal's maintenance condition is $fuel > distance(depot)$.

In total, using a reactive-behaviour maintenance goal, the rover has consumed 40 units of fuel in the process - 5 units moving towards the goal, 15 units moving back to the depot, and then 20 units moving towards the goal (refer to Figures 3.2 and 3.4).

Now, we consider this scenario using proactive-behaviour maintenance goals. The initial conditions are the same, however this time, the maintenance goals will be proactive.

In this case, the rover has consumed 30 units of fuel, saving 10 units. It has moved directly from location 10 to the depot, and then from the depot to the goal (See Figures 3.3

```

1  Rover is located at location 10. It has 20 units of fuel.
2  The maintenance condition is satisfied, so the rover moves towards location 20.
3
4  Rover is located at location 11. It has 19 units of fuel.
5  The maintenance condition is satisfied, so the rover moves towards location 20.
6
7  Rover is located at location 12. It has 18 units of fuel.
8  The maintenance condition is satisfied, so the rover moves towards location 20.
9
10 Rover is located at location 13. It has 17 units of fuel.
11 The maintenance condition is satisfied, so the rover moves towards location 20.
12
13 Rover is located at location 14. It has 16 units of fuel.
14 The maintenance condition is satisfied, so the rover moves towards location 20.
15
16 Rover is located at location 15. It has 15 units of fuel.
17 The maintenance condition is not satisfied, so the rover must refuel.
18
19 Rover is located at location 14. It has 14 units of fuel.
20 The maintenance condition is not satisfied, so the rover must refuel.
21
22 Rover is located at location 13. It has 12 units of fuel.
23 The maintenance condition is not satisfied, so the rover must refuel.
24
25 ...
26
27 Rover is located at location 0. It has 0 units of fuel.
28 The maintenance condition is not satisfied, so the rover must refuel.
29
30 Rover refuels.
31
32 Rover is located at location 0. It has 100 units of fuel.
33 The maintenance condition is satisfied, so the rover moves towards location 20.
34
35 The rover then takes 20 steps to the goal.

```

Figure 3.2: Step-by-Step Reactive Agent Example

and 3.5). This represents a 25% reduction in the fuel usage when compared with an agent behaving reactively.

3.1.2 Robot Soccer

Robotic sports events, such as RoboCup, have provided research prospects in highly dynamic environments. Not only must an agent consider adversarial robots in its plans, but its own actions are often unreliable – attempting to intercept a ball may cause it to bounce or roll away in an apparent random manner. The use of agent techniques, such as the BDI architecture, have been used to control teams of robot soccer players.

In this case study, we consider a robot playing the position of a defender in a game of robot soccer. Its general behaviour is to stay on the defensive side of the field and prevent


```

1 Rover is located at location 10. It has 20 units of fuel.
2 Before adopting the goal to move to location 20, it checks if its maintenance
3 condition will hold once it arrives.
4 It takes 10 units of fuel to get there, the rover is left with 10 units of fuel.
5 The goal is located 20 units away from the depot, therefore, the maintenance
6 condition will not hold.
7
8 The rover adopts the preventative goal. It still has 20 units of fuel.
9
10 The rover moves to the depot and refuels. It has 100 units of fuel.
11
12 The rover moves to the goal location.

```

Figure 3.3: Step-by-Step Proactive-behaviour Agent Example

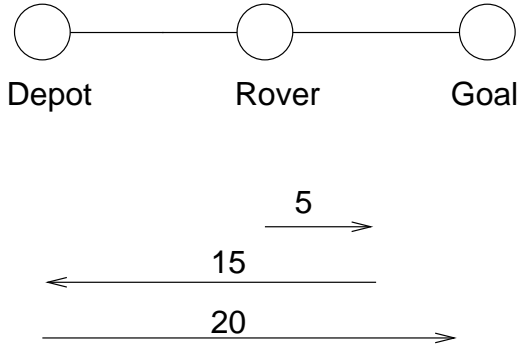


Figure 3.4: Scenario with reactive behaviour

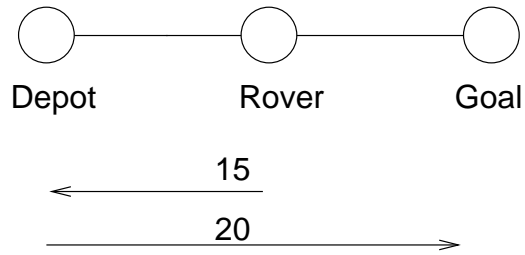


Figure 3.5: Scenario with proactive behaviour

the opposition from scoring.

One possible maintenance condition for such a defensive robot may be to ensure that opponent's forward robot never gets the ball. With the Mars rover, the actions to be taken in the case the maintenance goal is violated, and the case where the maintenance goal will be violated, are the same. In the robot soccer case, these actions are quite different. If the maintenance condition has not been violated, then the objective of the defender robot is to get into a position between the ball and the forward, making it difficult for the attacker to receive the ball. However, if the maintenance condition has been violated, the attacker has possession of the ball. In order to re-establish the maintenance condition, the defender robot should move to tackle the attacker, to steal the ball away.

In this case study, we establish that two separate, different behaviours may be necessary for the maintenance goal. In the first case, when the maintenance condition has been violated, the agent needs to act to re-establish the condition. The action the agent performs here is the

recovery goal, and closely models what occurs with reactive behaviour maintenance goals. In the second case, when the agent can predict that the maintenance condition may be broken, the agent performs a *preventative goal*, with the objective of disallowing the maintenance condition from becoming violated.

This maintenance goal may not be practical all the time however. It may be possible that a maintenance goal is not useful, and may be dropped. For example, if the opposing team’s strategy involved not have any robots forward, instead selecting to take ‘pot-shots’ from their own side of the field. In this case, the maintenance goal is not of use, and should be dropped. This illustrates the *failure condition* of a maintenance goal. When this condition is satisfied, it indicates that the maintenance goal serves no practical purpose or that the condition it is maintaining is unable to be maintained. In a similar fashion, a maintenance goal may also require a *enabling condition*. This is used to indicate when a maintenance goal should be considered by an agent. In this example, if the opposing team changed tactics from taking ‘pot-shots’ to having forwards, it is rational to readopt this maintenance goal to make the robots act as defenders once again.

3.2 Representation

In this section, we describe specific information of the various goal types discussed in this thesis. In particular, we define various goals types in terms of the attributes they consist of, and provide reasonings behind these.

We focus on achievement and maintenance goals, with a focus on showing how reactive and proactive behaviour in maintenance goals can be derived from the denoted attributes. Algorithms which utilise these representations will follow in the proceeding section.

3.2.1 Achievement Goals

Achievement goals are goals that have a specific state that the agent is attempting to bring about. This *state* is referred to as the *success* condition. When this state/success condition is realised, the achievement goal is dropped. Note that this can occur at any time while this goal is active, and may occur regardless as to the state of any plan that is realising this goal. A summary of these attributes can be found in Figure 3.6.

The success condition indicates when a goal has been achieved. This decouples the success of the goal from the success of the plan – that is, the agent can check for successful completion of the goal irrespective of the state of the plan. In the same way, the failure condition indicates

Goal Name	A unique identifier which acts as a handle to this goal.
Adopt Condition	When this condition is met, the agent may choose to pursue this goal.
Success Condition	A condition indicating the successful achievement of this goal.
Failure Condition	A condition that indicates that the achievement goal should be dropped.

Figure 3.6: Achievement Goal Definition

when the goal fails, not when the plan fails. Therefore, if a plan is executed and completes, but the goal's success condition is not met, the plan may be retried or a new plan attempted in its place. In the same manner, if a new plan is attempted, which fails during execution, rather than abandoning the goal, the goal persists and a new plan is attempted in its place.

3.2.2 Plans

Plans are mechanism by which a goal may be realised. These describe the sequence of actions that should be executed, in order to reach a particular goal state.

There are two main approaches to the concept of plans. The notion of a *plan-library* is common in many BDI systems, and involves a database of plans, linked to their associated goals. Therefore, when an agent adopts a goal, it refers to its database to determine the applicable plans that resolve these goals.

An alternative to the plan library is to use some planning mechanism, such as first principles planning, or the use of Hierarchical Task Networks. These approaches are beyond the scope of this thesis.

We outline the core attributes for *Plans* in Section 3.7. This lists the key attributes we require for our approach, and could be extended for other purposes if necessary.

The *Plan Body* is composed of elements from the agent's *Plan language*. Any suitable representation could be applicable at this point.

The Resources attribute consists of a tuple representing the resources required for this plan. In the case of a plan library, it is expected that a programmer provide these details. However, in cases of on-line planning, additional features may be required by the planner in order to be able to determine the resources involved.

The necessary and potential resources for each achievement goal can be determined from the set of all plans that realise this goal. The resources of each of these plans can be combined

Plan Name	A unique identifier which allows goals to refer to this plan.
Plan Body	A number of steps which, when followed, may resolve some condition. Plan steps may consist of primitive actions, sub-goals (which in turn, may call plans), and other operators such as sequence, parallel, and disjunction.
Pre-Conditions	The pre-conditions of a plan indicate the conditions that must be met before this plan can be executed.
Resources	The resource of a plan are the resource produced and consumed by all the primitive actions in this plan's plan body. It is used as a guide to determine the total consumption and production of resources as a result of executing this plan.

Figure 3.7: Plan Definition

in order to determine the *necessary* resources (the minimal set of resources common to all plans for this goal) and the *potential* resources (the maximal set of resources) for a particular goal.

Thangarajah et al. [2002b] provide more comprehensive discussion of resource summaries.

3.2.3 Maintenance Goals

Utilising the concepts outlined in the previous section, we now discuss the features consistent with maintenance goals. The attributes associated with maintenance goals follows in Figure 3.8.

Name	A unique name for referring to this maintenance goal.
Maintain Condition	The condition that an agent aims to keep satisfied while this goal is adopted.
Enable Condition	Once this condition has been satisfied, the agent should consider the maintenance goal during future deliberation.
Failure Condition	The failure condition indicates when a maintenance goal should no longer be pursued as part of the agent system, but should instead, be dropped.
Recovery Goal	The achievement goal that should be pursued in order to restore the maintenance condition, if it no longer holds.
Preventative Goal	The achievement goal that should be pursued in order to <i>prevent</i> the maintenance condition from becoming violated.

Figure 3.8: Maintenance Goal Definition

The recovery goal is an achievement goal, with the recovering or restoring the maintenance condition in the event that it no longer holds. It has its own plans, success and failure

conditions.

The preventative goal is an achievement goal, with the purpose of *preventing* a maintenance condition from becoming violated. When activated, it should perform actions that prevent the maintenance condition from being violated, for example, acquiring additional resources, modifying the environment, etc.

It is possible that either one of these goals could be absent. The presence of these goals imposes limitations on the types of behaviour possible for the maintenance goal. If the recovery goal is not present, then it is impossible for the maintenance goal to behave reactively, as it has no actions to respond with when the maintenance condition is violated. Similarly, if the preventative goal is absent, the maintenance goal cannot behave proactively, as it has no actions that can prevent failure.

If both goals are present, then the maintenance goal can act both proactively or reactively. If neither goal is present, then there are no actions available to support either behaviour. Instead, it may be possible for deliberation to be performed that considers the presence of the maintenance goal, and not adopt other goals that cause this maintenance goal's maintenance condition to fail. In this way, the behaviour of the maintenance goal without recovery or preventative goals is more akin to the behaviour associated with a constraint, that is, prevention of some *other* goal.

3.3 Algorithms

In this section, we will discuss how the previously defined representation enables the maintenance goal behaviour we have discussed earlier in this chapter. We will initially discuss common activities, such as how maintenance goals are adopted and how they may be dropped. We will then discuss separately, how this representation allows for both reactive and proactive behaviour.

Each goal has a number of *states* in which it may be. Events occur that cause the goals to transition between these states.

A maintenance goal, like an achievement goal, begins in a *pending state*. In this state, it does not influence the behaviour of an agent. It may exist in this state because it may conflict with other goals the agent currently has active, or simply because the agent has chosen not to activate it.

A maintenance goal enters the *maintaining state* once its *adopt condition* is satisfied. A maintenance goal in this state may influence the behaviour of the agent, in that the agent

should now monitor this goals maintenance condition.

The *failure condition* is used to indicate when an agent could ‘give-up’ on a goal, and allow it to be dropped. Once dropped, the goal is no longer part of the agent system, and may be used to indicate when some goal can no longer be maintained, or deemed no longer useful to consider by the agent.

In the interim, after the goal has been adopted and before the goal has failed, the agent will monitor the maintain condition of the maintenance goal. The recovery goal will be activated if the maintain condition is ever violated, and the preventative goal will be activated if the agent determines the maintain condition will be violated. The following sections describe how these processes work in more detail.

3.3.1 Reactive Behaviour

To exhibit reactive behaviour, an agent must continually monitor the maintenance condition of its maintenance goals. This could involve checking maintenance conditions every ‘*tick*’ or cycle through the agent interpreter (refer to Section 2.4), or at some frequent interval. In the event that a maintenance condition is not satisfied, the maintenance goal activates the *recovery goal* associated with this maintenance goal via the *recovery goal* attribute.

In activating this recovery goal, other goals the agent is currently pursuing may be forced to change to alternate states. For example, if a recovery goal was activated that involved moving the mars rover to a new location, other goals that moved the rover would have to be suspended until this recovery goal was satisfied. This process of resolving goal conflict has been discussed elsewhere (refer to Section 2.3).

3.3.2 Proactive Behaviour

For proactive behaviour, an agent must have some means of *predicting* the consequences of its actions. We introduce a new check, called **proactive-check**. Given the current beliefs of an agent, and the goals and plans it is currently pursuing, it is possible to *predict* the outcome of executing these goals. In practice, this prediction may not be perfect all of the time, due to dynamic changes in the environment, and the potential inability to perfectly predict the agent’s choices.

One possible method of achieving this is through the use of heuristics. One such heuristic may be to utilise *resource summaries*. Given a plan, it is possible to determine the resources that will be consumed and generated as a result of executing this plan.

The **proactive-check** construct accesses the current beliefs of an agent, as well as the plans the agent is currently pursuing. From the plans, the sum of the resources consumed can be found, and then compared with the resources the agent believes it has available to it. In the event that the agent has ample resources, it should continue its course of action, as it should not expect its maintenance conditions to be violated as a result of the execution of its current plan set.

However, if there are insufficient resources available, the agent should not continue its current course of action – doing so would lead to the eventual triggering of one or more of its maintenance goals. Instead, the agent will activate the *preventative goal* in the future-violated maintenance goal(s), and perform some plan that resolves this future violation of its maintenance conditions. Having done this, the agent can pursue its original goal as there is no potential imminent violation of any maintenance goals.

The algorithm for calculating which maintenance goals are satisfied after a set of achievement goals is completed is shown in Figure 3.9. The arguments to this function are the achievement goal to adopt (G_a) and the agent’s current resources (R). We assume that resources can be stored as $(name, value)$ pairs – for example, $(fuel, 100)$ may indicate the agent currently possesses 100 units of fuel, and $(weight, 10)$ may indicate that the agent is currently holding a total of 10kg of weight. For simplicity, we are using integer values, but this could be extended if required.

The **proactive-check** utilises the resource summary data in the following manner. The **proactive-check** function requires the new achievement goal to consider adopting (G_a), the currently available resources (R) and a means of determining the resource consumed by the active goals of the agent ($G_a.resourcesummaries$). NA' and PA' represent a list of the necessary and possible resources required respectively, for example $PA' = \langle fuel, 30 \rangle$, $NA' = \langle fuel, 10 \rangle$ indicates that 10 units of fuel will necessarily be consumed, and potentially, 30 units of fuel may be consumed. G_m is a list of all the maintenance goals the agent has active currently.

We also assume that it is possible to determine if a maintenance condition (mc) holds in a given environment. This environment can be constructed by taking the agent’s current resources, R , and applying either NA' or PA' , to determine the resources that are available. mc is some form of expression over the agent’s resources and beliefs, such as $fuel > 10$. We say that mc is consistent with R' if the maintenance condition evaluates to true, given the available resources.

The **proactive-check** function returns one of three results; *consistent*, *inconsistent* and

```

1 function proactive-check(Ga, R)
2   <NA', PA'> = Ga.resource summaries
3   number of maintenance goals := 0
4   number of maintenance goals satisfied := 0
5   number of maintenance goals unsatisfied := 0
6   CFG := <>
7
8   for each maintenance goal mg in Gm
9     mc = mg.maintenance condition
10    if mc is consistent with R - PA'
11      number of maintenance goals satisfied ++
12    if mc is not consistent with R - NA'
13      number of maintenance goals unsatisfied ++
14      CFG := < CFG + mg >
15      number of maintenance goals ++
16
17   if number of maintenance goals satisfied == number of maintenance goals
18     // all maintenance goals are definitely consistent
19     return consistent
20   else if number of maintenance goals unsatisfied > 0
21     // some maintenance goals are definitely unsatisfied
22     return <inconsistent, CFG>
23   else
24     return uncertain

```

Figure 3.9: The proactive-check algorithm

uncertain.

In that case where the achievement goal is consistent, the goal G_a is adopted and the agent execution cycle continues as normal. However, in the case where the **proactive-check** indicates that G_a is inconsistent, the preventative goal for each maintenance goal that is violated by G_a is adopted first.

In the case where **proactive-check** is uncertain, it is left to the agent developer to determine the most appropriate course of action. For example, a *bold* agent (as described in Section 2.2) may adopt the goal, risking it causing some maintenance goal to become violated. The reactive maintenance goal could then be triggered at some point in the future. Alternatively, if the agent was *cautious*, it may elect to adopt the preventative goals before pursuing G_a , even though the preventative goals may not be necessary.

For simplification, in the remainder of the thesis, we will assume that the result from the **proactive-check** function is either consistent or inconsistent; in cases where future is uncertain, we adopt the preventative goal (where possible). Naturally, any other mechanism could be used in this place to make an alternate course of action concerning uncertain futures.

This algorithm can be added easily to the abstract agent interpreter illustrated in Figure 2.4. In the step 2 deliberative phase, decisions will be made concerning whether or not the

maintenance goals in the agent system will be activated. If so, the deliberative phase may also need to suspend currently active goals.

It is possible that both reactive and proactive maintenance goals can be implemented using the same algorithms, provided that the proactive version supports the notion of the **proactive-check** construct described earlier. In the following chapter, we will discuss in more detail how this is possible.

3.3.3 Mars Rover Revisited

We return to our case studies to offer an operational example of the representation and algorithms. This example will consist of two parts – operation with and without proactive maintenance goals. In this way, we aim to contrast the behaviour when this goal type is available to the agent.

The Mars rover moves about some environment, consuming fuel as it moves, at the rate of 1 unit of fuel consumed for every 1 unit of distance moved. It carries a limited supply, but a depot is present (at location (0,0)) where it can refill its tank to full capacity. In this case study, let us assume that the rover begins at location 10 with 20 units of fuel remaining.

An achievement goal in this scenario is to move the rover to location 20. Let us assume that the agent can move 1 unit at a time – therefore, this requires the agent to move to location 11, then location 12, and so on until it reaches location 20. This means that in this case, a *plan* for this achievement goal is `move11;move12;...;move19;move20`.

Overall, the achievement goal may be represented with the data structure outlined in Figure 3.10.

Goal Name	MoveTo20Goal
Success Condition	<i>location</i> = (20,0)
Failure Condition	<i>location</i> ≠ (0,0) and fuel = 0
Plans	<move11;move12;...;move19;move20>

Figure 3.10: MoveTo20Goal Specifications

The goal succeeds when the location of the rover is (20,0). However, if it ever reaches a point where the location is not (0,0) and it has no fuel, this goal should be dropped – it is impossible for the rover to move any further, so it no longer has any means for realising this goal.

In this situation, we have stored the suitable plan in the *Plans* attribute. In practice however, it is possible that some form of planning will be necessary to generate the appro-

priate actions for this goal. Our representation and algorithms support such mechanisms, with a slightly altered representation. We therefore support features such as lookups in a plan library and first-order or HTN-style planning.

A suitable maintenance goal for this system is to make sure an agent always has sufficient fuel to return to the depot. For reasons discussed earlier, this is preferred to a maintenance condition where the rover’s fuel level is compared with some constant value.

There are two possible behaviours for this maintenance goal – the reactive and the proactive behaviour. We will treat this maintenance goal as two separate maintenance goals, each describing a single behaviour. Splitting the maintenance goal into two separate behaviours is done for convenience, to allow easier comparisons to be made in later work. Conceptually, these behaviours correspond to the same maintenance goal.

The maintenance condition for this goal is to ensure that there is always enough fuel for the rover to return to the depot where it may refuel. If this is ever the case, this reactive behaviour should be performed, resulting in the agent refuelling – successful accomplishment of this occurs when $\text{fuel} = 100$. The maintenance goal is said to fail if it ever runs out of fuel, and it is not located at position (0,0).

This maintenance goal should always be active, except in the case where there is a meteor storm. If there is a meteor storm, this goal should no longer be considered. A summary of the maintenance goal specification can be found in Figure 3.11.

Maintain Condition	<i>fuel > distance(depot)</i>
Enable Condition	<i>not meteor shower</i>
Failure Condition	<i>meteor shower</i>
Recovery Goal	RefuelRoverGoal

Figure 3.11: Example Reactive Maintenance Goal Specification

With regards to the proactive behaviour component, a similar structure is used. The important difference is that the maintenance condition considers the effects of the agent’s current goals to determine if the maintenance goal will be violated *in the future*. To this end, we use the *proactive-check* construct described earlier. A summary of this maintenance goal specification (with proactiveness) can be found in Figure 3.12.

We now illustrate how these goals operate in a Mars rover setting. In the first example, only the reactive behaviour will be used, while in the second example, both reactive and proactive behaviours (recovery and preventative goals) are used.

Maintain Condition	<i>proactive – check(fuel < distance(depot))</i>
Enable Condition	<i>not meteor shower</i>
Failure Condition	<i>meteor shower</i>
Recovery Goal	RefuelRoverGoal
Preventative Goal	RefuelRoverGoal

Figure 3.12: Example Maintenance Goal Specification utilising **proactive-check**

Reactive Behaviour Only

Beginning with the rover in location 10 with 20 units of fuel, the agent only has the reactive maintenance goal adopted. This example begins with the agent adopting the achievement goal of moving to location (20,0).

Adopting the moveTo20 Goal, a suitable plan is selected or generated. In this case, this plan is `moveTo11;moveTo12;...;moveTo19;moveTo20`.

Before executing any actions, the rover tests the maintenance conditions of its maintenance goals. Being located at location 10 with 20 units of fuel, the maintenance goal is satisfied, so execution may continue.

Execution of this plan begins, with the rover performing the first step of the achievement goal's plan, `move11`. This results in the state being the rover is now located at location (11,0), with 19 units of fuel.

Repeated execution will result in the rover eventually being located at location (15,0) with 15 units of fuel remaining. When the agent checks the maintenance condition, `fuel > distanceToDepot`, on this occasion, it fails. The recovery goal is activated, which causes the achievement goal to be suspended. We assume that the preventative goal's plan is `(moveToDepot;refuel)`, which is then pursued, resulting eventually in the rover being positioned at location (0,0) with 100 units of fuel after refuelling. The achievement goal is then resumed, and this time completes uninterrupted.

Reactive and Proactive Behaviour

We begin this example with the same initial conditions as before, the rover located at location (10,0) with 20 units of fuel. It only has maintenance goal adopted and currently maintaining, and the example begins with the adoption of the achievement goal, `moveTo20`.

Adopting the achievement goal, a suitable plan is selected or generated. In this case (as before), this plan is `move11;move12;...;move19;move20`. Before execution begins, the agent must determine if the maintenance condition for the proactive maintenance goal will

hold at the conclusion of the achievement goal.

Moving from the current location to location (10,0) will consume 10 units of fuel. Given that the rover has 20 units of fuel initially, this will leave 10 units of fuel remaining after execution of the `moveTo20`. At this time, it will be located at location (20,0) with 10 units of fuel remaining, hence the maintenance condition, $fuel > distanceToDepot$ will not hold. Therefore, the preventative goal is activated. Using existing goal conflict resolution strategies, the achievement goal is suspended until the preventative goal is satisfied, which it will be once the success condition of $fuel = 100$ is obtained. In order to do so, a plan is generated, which in this case, is `moveToDepot;refuel`.

After this plan is executed, the rover is located at location (0,0) with 100 units of fuel. The achievement goal, `moveTo20` is reactivated after the preventative goal is completed. Pursuing the achievement goal this time is successful and uninterrupted.

3.3.4 Potential Optimisations

In some environments, continually checking for the states of maintenance conditions may be avoided. In the case of static environments, we only need to perform a check on the future state of maintenance conditions when new goals are added to the agent system. As the environment does not change, the only change is the actions the agent will be performing, which is determined by the other goals in the agent system, and the plans that been selected for it to execute.

It is also important to note that if the prediction model employed by the agent is perfect, the recovery goal will never be pursued (assuming that there is a proactive check made for the same maintenance condition). As the proactive check is always correct and no unexpected changes in the environment are possible (and all plans succeed as expected), it will always detect future violation of its maintenance conditions before the recovery goal is required. In practice, this is unlikely however. This will be investigated further in Chapter 5.

Summary

In this chapter, we have presented a suitable representation of maintenance goals which capture the desired behaviour that were outlined via earlier case studies. The two relevant behaviours, reactive and proactive, have been conceptually illustrated as two separate goals. In practice, one could design a solitary maintenance goal that exhibits both behaviours. We have shown how this representation models both the reactive and proactive behaviours

discussed in this chapter. In the following chapter, we will present a formal description of this representation and algorithms, and illustrate potential optimisations.

Chapter 4

Operational Semantics

In previous chapters, we identified the characteristics we desire in maintenance goals, both reactive and proactive. To support proactive behaviour that fulfilled our requirements, maintenance goals required a maintenance condition that the agent aims to keep satisfied, enabling and disabling conditions, to activate and suspend the maintenance goal, and a recovery goal and a preventative goal, which were activated at the appropriate times.

In this chapter, we provide formal semantics for proactive maintenance goals that support the characteristics and behaviours developed in this thesis. We begin by introducing and discussing an existing formal model of goals in agent systems, which we further develop and expand to incorporate maintenance goals, both reactive and proactive. We then detail the changes we make to this existing framework.

Finally, we conclude this chapter by discussing maintenance goals in the context of this newly developed formalism, proving certain desirable features and showing how the formalism ensures this.

4.1 Issues for Formalisation

As described in Section 3.2.3, there are specific requirements as to the behaviour of proactive maintenance goals. Any operational semantics developed that support proactive maintenance goals should include the attributes as described in Figure 3.8.

In the remainder of this chapter, we will develop formal semantics which support these requirements. We will develop formal semantics by extending existing notions of formal semantics, originally developed by van Riemsdijk et al. [2008], and then illustrate these semantics in operation by returning to our Mars rover case study.

4.2 Maintenance Goal Formalisation

The original formalism of van Riemsdijk et al. [2008] describes the generic attributes shared by all goal types. There is limited discussion of how beliefs operate and how actions towards achieving goals are performed, which may be crucial for a complete agent system. Overall, the state of an agent is represented by the agent's belief-base (B) and goal-base(G), in the structure $\langle B, G \rangle$.

All goals have a state attribute. There are two possible states that a goal can have, *suspended* or *active*. A suspended goal identifies one that is not applicable at the current point in time, while an active goal is applicable.

Goals can transition between states if a particular *condition* arises. Each possible transition that can occur to a goal is defined in the form of a *condition/action* pair. A condition is a test to determine if some particular expression holds according to the current beliefs of the agent. If this condition is met, the *action* is then applicable. The formalism identifies three possible actions that may be applied to a goal if this condition is satisfied, *activate*, *suspend* or *drop*. Activating a goal transitions the goal to the active state, suspending a goal transitions the goal to the suspended state, and dropping the goal removes this goal from the agent's goal-set.

Some examples of condition/action pairs are (*fuel* < 10, *suspend*), (*true*, *activate*) and (*location* == (10,10), *drop*). In the first pair, if the agent believes that the level of fuel is less than 10, the goal associated with this pair should be suspended. In the second example, the goal may be made active at any time, due to the condition being *true* (which we assume, always succeeds). Lastly, the goal in the final example may be dropped if the agent believes its location is (10,10). When a goal is dropped, it is removed from the agent's goal-set and is no longer considered in the agent's deliberation process. Reasons for dropping a goal including the successful completion of the goal, and also if it is no longer possible to achieve the goal.

A goal also includes π , which acts as a placeholder for a plan. As discussed earlier, there is no specific mechanism for the representation of a plan in this framework. The primary requirement is that it must be possible to execute π in a step-by-step manner, so that executing a step in π will produce π' , which is also executable. This allows this platform to support agents that utilise a plan library, as well as those that plan from first principles. Plans are supplied to goals via a *means-ends-reasoning* (MER) function.

Each goal may have any number of condition/action pairs associated with it, storing these

as a set of pairs. There are two sets associated with a goal, set C and E . Condition/action pairs stored in set C are applicable when π is not empty, for example, if a goal is suspended (transitions from active to suspended), it may be possible to resume it by executing the remainder of the plan found in π . The condition/action pairs stored in set E are applicable when π is empty, for example, after a goal initially transitions from the suspended to the active state.

In summary, a goal has the form $g(C,E,S,\pi)$, where C and E are set of *condition/action* pairs, S is the *state* of the goal and π acts as a placeholder for a plan.

4.2.1 Operation

The goal-base of the agent begins with a number of goals, in the form $g(C,E)$. These goals do not have a notion of state initially. The formalism described here and in Figure 4.2 illustrates the operation and execution for a single goal, which may then be applied across all the goals in the agent's goal-base (Rule 1 in Figure 4.2).

Once adopted, goals begin in the suspended state, with π being empty, $g(C,E,\text{Suspended},\epsilon)$ (Rule 2 from Figure 4.2). When a goal is in the suspended state, only condition/action pairs where the action is *activate* are considered. This means that goals may not be dropped from the suspended state. As π is empty, only pairs in set E are considered. If one of the conditions is satisfied by the agent's beliefs, the goal transitions to the active state. If π was not empty, then the condition/action pairs from set C would have been considered instead (Rules 3 and 4 from Figure 4.2). This can occur when a goal has been attempted, but a particular action has occurred which has moved that goal to the suspended state.

When a goal is in the active state, condition/action pairs where the action is either *suspend* or *drop* are considered. Again, if π in this goal is empty, the pairs from set E are tested, while if π is not empty, the condition/actions pairs from set C are tested (Rules 5—8 from Figure 4.2).

Figure 4.1 illustrates the goal life-cycle utilised in these formal semantics. A goal begins in the suspended state, may transition between the active and suspended states, before finally being dropped from the active state.

If π is empty for an active goal, MER is used to generate an appropriate plan for this goal (Rule 9 in Figure 4.2). If an agent attempts to perform a plan step which fails, the plan in π is discarded. MER can then generate a new plan for this goal, assuming no other conditions arise that cause the goal to be suspended or dropped (Rule 11 in Figure 4.2). Otherwise,

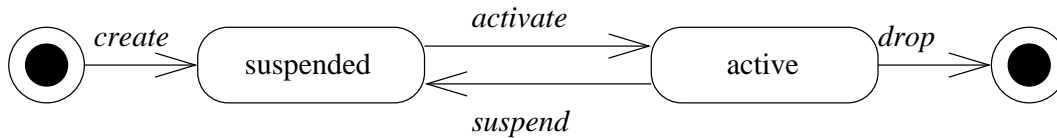


Figure 4.1: Goal Life-cycle from van Riemsdijk et al. [2008]

a plan step involves executing a single step in π , provided no other rules are applicable for that goal at that time (Rule 10 in Figure 4.2).

In summary, the rules listed in Figure 4.2 describe the behaviour of agents utilising this framework.

4.3 Complete State Transition Rules

The previous formalism has several limitations that prevent support for maintenance goals with proactive behaviour. The most apparent is the need for a third state in which a goal may exist, in addition to **suspended** and **active**. This is due to maintenance goals requiring 3 states – one when they are inactive and do not influence the agent, one when they are active, but not performing any actions, and a third state for when a maintenance goal is causing actions to be performed. For proactive maintenance goals, the framework also requires a mechanism by which prediction can be incorporated.

We have modified the operational semantics of van Riemsdijk et al. [2008] to include the characteristics of maintenance goals found in Section 4.1, to produce the formal semantics listed in Figure 4.3. The explicit notion of goal-condition, GC , has been added to all goals from the original formalism, while rules 12 thru 17 have been added to include the transitions to and from the maintaining state. Importantly, rule 9 has been modified to provide the goal-condition to the MER function. For simplicity, we have combined the sets C and E into a single set, CAP , which contains all condition/action pairs. We have extended the condition to include notions such as the current state of each goal, hence this information can be included in the condition, which eliminates the need for separate sets for when the goal is suspended or active.

We will now compare the framework by van Riemsdijk et al. [2008] with our modified framework that supports maintenance goals with proactive behaviour.

Rule 1 and rule 2 remain the same in both frameworks, which are the rules that allow an

$$\begin{array}{c}
\frac{\langle B, g \rangle \longrightarrow \langle B', g' \rangle \quad g \in G \quad G' = (G \cup \{g'\}) \setminus \{g\}}{\langle B, G \rangle \longrightarrow \langle B', G' \rangle} \quad 1 \\
\\
\frac{}{\langle B, g(C, E) \rangle \longrightarrow \langle B, g(C, E, \text{Suspended}, \epsilon) \rangle} \quad 2 \\
\\
\frac{\pi \neq \epsilon \quad \langle c, \text{Activate} \rangle \in C \quad B \models c}{\langle B, g(C, E, \text{Suspended}, \pi) \rangle \longrightarrow \langle B, g(C, E, \text{Active}, \pi) \rangle} \quad 3 \\
\\
\frac{\langle c, \text{Activate} \rangle \in E \quad B \models c}{\langle B, g(C, E, \text{Suspended}, \epsilon) \rangle \longrightarrow \langle B, g(C, E, \text{Active}, \epsilon) \rangle} \quad 4 \\
\\
\frac{\pi \neq \epsilon \quad \langle c, \text{Suspend} \rangle \in C \quad B \models c}{\langle B, g(C, E, \text{Active}, \pi) \rangle \longrightarrow \langle B, g(C, E, \text{Suspended}, \pi) \rangle} \quad 5 \\
\\
\frac{\langle c, \text{Suspend} \rangle \in E \quad B \models c}{\langle B, g(C, E, \text{Active}, \epsilon) \rangle \longrightarrow \langle B, g(C, E, \text{Suspended}, \epsilon) \rangle} \quad 6 \\
\\
\frac{\pi \neq \epsilon \quad \langle g(C, E, \text{Active}, \pi) \rangle \in G \quad \langle c, \text{Drop} \rangle \in C \quad B \models c}{\langle B, G \rangle \longrightarrow \langle B, G \setminus \{g(C, E, \text{Active}, \pi)\} \rangle} \quad 7 \\
\\
\frac{\langle g(C, E, \text{Active}, \epsilon) \rangle \in G \quad \langle c, \text{Drop} \rangle \in E \quad B \models c}{\langle B, G \rangle \longrightarrow \langle B, G \setminus \{g(C, E, \text{Active}, \epsilon)\} \rangle} \quad 8 \\
\\
\frac{\neg \exists \langle c, a \rangle \in E \cdot (B \models c) \wedge (a \neq \text{Activate})}{\langle B, g(C, E, \text{Active}, \epsilon) \rangle \longrightarrow \langle B, g(C, E, \text{Active}, \text{mer}(g, B)) \rangle} \quad 9 \\
\\
\frac{\langle B, \pi \rangle \longrightarrow \langle B', \pi' \rangle \quad \neg \exists \langle c, a \rangle \in C \cdot (B \models c) \wedge (a \neq \text{Activate})}{\langle B, g(C, E, \text{Active}, \pi) \rangle \longrightarrow \langle B, g(C, E, \text{Active}, \pi') \rangle} \quad 10 \\
\\
\frac{\pi \neq \epsilon \quad \langle B, \pi \rangle \nrightarrow \langle B', \pi' \rangle}{\langle B, g(C, E, \text{Active}, \pi) \rangle \longrightarrow \langle B, g(C, E, \text{Active}, \epsilon) \rangle} \quad 11
\end{array}$$

Figure 4.2: Formal Semantics from van Riemsdijk et al.'s framework

$$\begin{array}{c}
\frac{\langle B, g \rangle \longrightarrow \langle B', g' \rangle \quad g \in G \quad G' = (G \cup \{g'\}) \setminus \{g\}}{\langle B, G \rangle \longrightarrow \langle B', G' \rangle} \quad 1 \\
\\
\frac{}{\langle B, g(\text{name}, GC, CAP) \rangle \longrightarrow \langle B, g(\text{name}, GC, CAP, \text{Pending}, \epsilon) \rangle} \quad 2 \\
\\
\frac{\langle c, \text{Activate} \rangle \in CAP \quad B \models c}{\langle B, G \cup g(\text{name}, GC, CAP, \text{Pending}, \epsilon) \rangle \longrightarrow \langle B, G' \cup g(\text{name}, GC, CAP, \text{Active}, \epsilon) \rangle} \quad 3 \\
\\
\frac{\langle c, \text{Maintain} \rangle \in CAP \quad B \models c}{\langle B, G \cup g(\text{name}, GC, CAP, \text{Pending}, \epsilon) \rangle \longrightarrow \langle B, G' \cup g(\text{name}, GC, CAP, \text{Maintaining}, \epsilon) \rangle} \quad 4 \\
\\
\frac{\langle c, \text{Pend} \rangle \in CAP \quad B \models c}{\langle B, G \cup g(\text{name}, GC, CAP, \text{Active}, \epsilon) \rangle \longrightarrow \langle B, G' \cup g(\text{name}, GC, CAP, \text{Pending}, \epsilon) \rangle} \quad 5 \\
\\
\frac{\langle c, \text{Maintain} \rangle \in CAP \quad B \models c}{\langle B, G \cup g(\text{name}, GC, CAP, \text{Active}, \epsilon) \rangle \longrightarrow \langle B, G' \cup g(\text{name}, GC, CAP, \text{Maintaining}, \epsilon) \rangle} \quad 6 \\
\\
\frac{\langle c, \text{Activate} \rangle \in CAP \quad B \models c}{\langle B, G \cup g(\text{name}, GC, CAP, \text{Maintaining}, \epsilon) \rangle \longrightarrow \langle B, G' \cup g(\text{name}, GC, CAP, \text{Active}, \epsilon) \rangle} \quad 7 \\
\\
\frac{\langle c, \text{Pend} \rangle \in CAP \quad B \models c}{\langle B, G \cup g(\text{name}, GC, CAP, \text{Maintaining}, \epsilon) \rangle \longrightarrow \langle B, G' \cup g(\text{name}, GC, CAP, \text{Pending}, \epsilon) \rangle} \quad 8 \\
\\
\frac{\langle c, \text{DROP} \rangle \in CAP \quad B \models c}{\langle B, G \cup g(\text{name}, GC, CAP, \text{Active}, \epsilon) \rangle \longrightarrow \langle B, G \rangle} \quad 9 \\
\\
\frac{\langle c, \text{DROP} \rangle \in CAP \quad B \models c}{\langle B, G \cup g(\text{name}, GC, CAP, \text{Pending}, \epsilon) \rangle \longrightarrow \langle B, G \rangle} \quad 10 \\
\\
\frac{\langle c, \text{DROP} \rangle \in CAP \quad B \models c}{\langle B, G \cup g(\text{name}, GC, CAP, \text{Maintaining}, \epsilon) \rangle \longrightarrow \langle B, G \rangle} \quad 11 \\
\\
\frac{\neg \exists \langle c, a \rangle \in E \cdot (B \models c) \wedge (a \neq \text{Activate}) \quad \Pi = \text{mer}(GC, B, G) \quad \Pi \neq \epsilon}{\langle B, G \cup g(GC, CAP, \text{Active}, \epsilon) \rangle \longrightarrow \langle B, G \cup g(GC, CAP, \text{Active}, \Pi) \rangle} \quad 12a \\
\\
\frac{\neg \exists \langle c, a \rangle \in E \cdot (B \models c) \wedge (a \neq \text{Activate}) \quad \Pi = \text{mer}(GC, B, G) \quad \Pi = \epsilon}{\langle B, G \cup g(GC, CAP, \text{Active}, \epsilon) \rangle \longrightarrow \langle B, G \rangle} \quad 12b \\
\\
\frac{\begin{array}{c} \langle B, \pi \rangle \longrightarrow \langle B', \pi' \rangle \\ \neg \exists \langle c, a \rangle \in C \cdot (B \models c) \wedge (a \neq \text{Activate}) \\ \forall G' \in MG(G) \neg \exists (c, \text{activate}) \in \text{rules}(G) B \models c \end{array}}{\langle B, G \cup g(GC, CAP, \text{Active}, \pi) \rangle \longrightarrow \langle B, G \cup g(GC, CAP, \text{Active}, \pi') \rangle} \quad 13 \\
\\
\frac{\pi \neq \epsilon \quad \langle B, \pi \rangle \rightarrow \langle B', \pi' \rangle}{\langle B, G \cup g(GC, CAP, \text{Active}, \pi) \rangle \longrightarrow \langle B, G \cup g(GC, CAP, \text{Active}, \epsilon) \rangle} \quad 14
\end{array}$$

Figure 4.3: Modified Operational Semantics

agent to initially adopt a goal, which defaults to the pending state, as well as updates to the agent's belief set.

In van Riemsdijk et al. [2008]'s framework, rules 3 through 6 allow an agent to transition between states when a suitable condition is met. There are four possible transitions, as there are two states in this system, and there are two locations for condition/action pairs, sets C and E , which are used to differentiate between goals with selected plans and those withouts.

In contrast to these rules, in our framework, state transitions are performed using rules 3 through 8. We have three states to handle, hence the need for an increased number of rules. The number of rules is reduced as we have combined the sets C and E into a single set of condition/action pairs, CAP .

Rules 7 and rules 8 of van Riemsdijk et al. [2008]'s framework allow an agent to drop a goal given a specified condition. One rule is required per state, and our modified framework supports dropping goals using rules 9, 10 and 11.

Rule 9 is used to determine a suitable plan for a goal, and allows mechanisms such as *means-ends-reasoning* or a plan-library to be employed. In our framework, this process consists of two rules, rule 12a and rule 12b. The reason for the separation is to allow for planning to fail. If a plan can be generated (or selected from the plan library), then the agent cycle continues as normal (Rule 12a). If no plan can be selected however (Rule 12b), the goal is dropped. Arguably, this is rational behaviour (to drop plans an agent has no means of achieving), however alternative approaches may exist, such as holding onto this goal until such a time as a plan is available.

In van Riemsdijk et al. [2008]'s framework, rules 10 and 11 allow an agent to perform plan steps. Rule 10 occurs when a plan step is successful, while rule 11 is processed when the plan step fails – in this case, eliminating the entire plan, which would subsequently cause rule 9 to be activated once again.

This process occurs in a similar fashion in our modified framework, corresponding to rule 13 and rule 14. The most significant change is that a plan step may not be executed if it would cause a maintenance goal to be violated.

Our modified framework simplifies some aspects of the original framework by van Riemsdijk et al. [2008], as well as addressing the limitations of maintenance goals. In our framework, we have decoupled the prediction aspect of maintenance goals from the actions associated with recovery and prevention. This is accomplished by causing maintenance goals to be activated when a condition from the CAP set is satisfied, as indicated in rule 13. This separation will be discussed further in the following section.

4.4 The future Operator

In our modified semantics, we have altered the behaviour of maintenance goals to more closely match the behaviour found in agent systems, such as Pokahr et al. [2005a]. There are three states a maintenance goal can exist in: *pending*, *maintaining* and *active*. Of these, the most important state for maintenance goals with proactive behaviour is the *maintaining* state. It is in this state that the maintenance goal proactively check for future violations of its maintenance condition, and if it will not hold in the future, transition to the *active* state and perform actions to prevent future violation.

Incorporating a continual, proactive checking mechanism into our framework involves checking the future status of maintenance conditions via the conditions in the CAP of the maintenance goals. To achieve this, we introduce a new operator, **future**, which can be used to check the status of maintenance conditions in the future.

There are many approaches to realising the **future** construct. In the following sections, we will describe three such mechanisms and discuss issues arising from their approaches.

4.4.1 Resource Summary based

As discussed in Section 2.5.1, resource summaries have been shown to be effective at reducing or eliminating negative interaction between (achievement) goals, as discussed by Thangarajah et al. [2002b].

A **future** construct utilising resource summaries could calculate the resources generated and consumed by other active (achievement) goals the agent currently has. In our framework, this information is stored in the goal set of the agent, G . The currently available resources is a belief, stored in B of the agent. Therefore, the **future** construct could appear as $future(mc, B, G)$, where mc is the maintenance condition to check. If the maintenance condition will not hold after all active goals in G are complete, based on the resource they will use and the current beliefs the agent has about its beliefs, then its reactive maintenance goal will be triggered at some point during the execution of all G . Therefore, to avoid the reactive maintenance goal from triggering, the agent should pursue the proactive maintenance goal first.

Cautious vs Bold Agents

There arises some issues concerning how an agent should behave in response to a prediction made by the **future** function. In our representation, **future** returns a boolean value, indicating

whether or not a maintenance condition holds given the agent’s current beliefs and intended goals. However, there is a third response that *future* could return: *uncertain*. This can occur when there are multiple ways in which a goal may be accomplished, and one of these plans does not violate the maintenance condition, while another does violate this condition. At this point in the process, the agent has no idea which plan will be selected, therefore *future* cannot predict with certainty if the maintenance condition will hold or not in the future.

There are two responses an agent can take given an uncertain future. The first is to proceed as normal and ‘hope-for-the-best’ concerning its maintenance goals. This is the *bold* approach (as discussed in Section 2.2), and can reduce how often the proactive maintenance goal is used, but can increase the frequency the reactive behaviour occurs.

The alternate course of action, given an uncertain future, is to be *cautious* (as described in Section 2.2) and adopt the maintenance goal’s preventative goal, thereby acting proactively. This will ensure that the recovery goal will not be called, as well as ensuring that the maintenance condition is never violated. This is especially important in cases of safety.

Our preliminary suggestion with respect to application of the *future* function is for agents to act *cautiously*. That is, that if the agent is not 100% certain that its maintenance conditions will hold, to employ the proactive behaviour and adopt the maintenance goal’s preventative goal. A more comprehensive investigation of reactive or proactive behaviour concerning maintenance goals will be performed in Chapter 5.

However, if an agent chooses to be bold, the reactive behaviour of the maintenance goal can offer a ‘backup’ if the *future* prediction is incorrect. This does lead to inefficiency however, so the reactive behaviour of the maintenance goal should be relied upon as little as possible.

4.4.2 User Supplied

One alternative to using resource summaries is to allow an agent developer to write their own *future* implementation. The framework remains unmodified, as the *future* function simply checks the given conditions and returns a boolean indicating if they hold or do not in the future.

This allows a user to select a mechanism most appropriate for the task at hand, and may include such techniques such as machine learning, neural networks or markov models. There is a large body of work that has the potential to be of use in this area, which we will cover in Chapter 6. This is by no means a simple task, hence the need for approximation and heuristic approaches.

4.5 Case Studies

We return to our example of the Mars rover, illustrating its behaviour with reference to the formal semantics. We first describe the process using only reactive behaviour, and then repeat the process with proactive proactive.

A maintenance goal with reactive behaviour to manage an agent's fuel can be represented as,

$$g(ref, fuel = 100, CAP, maintaining, \pi),$$

where CAP consists of the following condition/action pairs.

$$\begin{aligned} &\langle ref.state = Maintaining \wedge fuel \leq distanceToDepot, ACTIVATE \rangle \\ &\langle ref.state = Active \wedge fuel == 100, MAINTAIN \rangle \end{aligned}$$

The first rule states that if the rover is ever located at a point where the distance to the depot is equal to or exceeds the fuel remaining in the tank, it should activate the maintenance goal. This will cause a plan to be selected for this goal, with the aim being to have a state where the fuel level is 100%.

The second rule states that if the rover was trying to refuel (i.e, this goal's state was **Active**), and the fuel level reached 100%, it should go back to **Maintaining**. This avoids the problem first expressed by Braubach et al. [2004] where a rover may only partially fill the tank due to satisfying the maintain condition.

An achievement goal in this example may be for the rover to move to location 6. This can be represented as the following goal.

$$g(to6, location = 6, CAP, pending, \pi)$$

where CAP consists of the following condition action pairs.

$$\begin{aligned} &(to6.state = Pending \wedge refuel.state \neq Active, ACTIVATE) \\ &(to6.state = Active \wedge refuel.state = Active, PEND) \\ &(to6.state = Active \wedge location = 6, DROP) \end{aligned}$$

In the first rule, the rover can activate this goal if the goal is **pending**, and if the refuel goal is not **Active**. If the refuel goal is **Active**, and we attempt to adopt the *to6* goal, the

goals may interfere with one another – hence, they are prevented from both being active simultaneously.

The second rule operates in a similar way, ensuring that if the refuel goal is adopted, the *to6* goal transitions to the pending state, again to prevent interference.

The final rule allows the agent to drop the *to6* goal once it is at location 6.

We show how the goal set of the agent evolves as it attempts to achieve the *to6* goal. As the goal condition and *CAP* remains static for the life of each goal, they will not be repeated here. We begin with the rover located in location 4 with 6 units of fuel in its tank. The maintenance goal starts in the **Maintaining** state, while the *to6* goal begins in the **pending** state. For clarity, we will also keep explicit track of the fuel level as $f(F)$, where F is the current fuel level. Hence we commence in the configuration below, where we write MG for $g(ref, \text{Maintaining}, \epsilon)$.

$$\{f(6), MG, g(to6, \text{Pending}, \epsilon)\}$$

As the maintain condition is not triggered, the *to6* goal is made active, a plan is found for it, and this plan commences execution.

$$\frac{\frac{\{f(6), MG, g(to6, \text{Pending}, \epsilon)\}}{\{f(6), MG, g(to6, \text{Active}, \epsilon)\}} 1}{\{f(6), MG, g(to6, \text{Active}, m3; m4; m5; m6)\}} 10a$$

$$\frac{}{\{f(4), MG, g(to6, \text{Active}, m5; m6)\}} 11 \times 2$$

At this point the fuel level is 4 with the rover at location 4. This means that the maintenance goal is activated, as the condition $fuel \leq distanceToDepot$ is now true. We then move the *to6* goal to **pending**, activate the maintenance goal and generate a plan for it.

$$\frac{\frac{\{f(4), g(ref, \text{Maintaining}, \epsilon), g(to6, \text{Active}, m5; m6)\}}{\{f(4), g(ref, \text{Active}, \epsilon), g(to6, \text{Active}, m5; m6)\}} 5}{\{f(4), g(ref, \text{Active}, \epsilon), g(to6, \text{Pending}, m5; m6)\}} 3$$

$$\frac{}{\{f(4), g(ref, \text{Active}, m30; refuel), g(to6, \text{Pending}, m5; m6)\}} 10a$$

where $m30$ is the sequence $m3; m2; m1; m0$. We now execute the plan for the maintenance goal, which refills the tank. The maintenance goal then returns to the **Maintaining** state and the original goal is reactivated.

$$\frac{\frac{\frac{\{f(4), g(ref, Active, m30; refuel), g(to6, Pending, m5; m6)\}}{\{f(100), g(ref, Active, \epsilon), g(to6, Pending, m5; m6)\}} 11^5}{\frac{\{f(100), MG, g(to6, Pending, m5; m6)\}}{\{f(100), MG, g(to6, Active, m5; m6)\}} 1} 4$$

At this point, the plan fails, as the rover is no longer at position 4. Hence the plan fails, and a new one is found, which then achieves the *to6* goal, which is then dropped.

$$\frac{\frac{\frac{\{f(100), MG, g(to6, Active, m5; m6)\}}{\{f(100), MG, g(to6, Active, \epsilon)\}} 12}{\frac{\{f(100), MG, g(to6, Active, m1; m2; m3; m4; m5; m6)\}}{\{f(94), MG, g(to6, Active, \epsilon)\}} 7} 10a$$

$$\frac{\{f(94), MG\}}{\{f(94), MG\}} 11^6$$

4.5.1 Example of Maintenance Goals with proactive behaviour

A maintenance goal with proactive behaviour has the same structure as any other goal in the goal set, that is, $g(name, goalcondition, CAP, state, \pi)$. Typically, *CAP* will include condition/action pairs that cause the maintenance goal to become **Active** when *future* predicts that its maintain condition does not hold, i.e. that *future*($\neg goalcondition$) holds. In these cases, the associated action is to **ACTIVATE** the maintenance goal.

We expect that achievement goals that should not run concurrently with this maintenance goal have a condition/action pair similar to the following

$$\langle thisgoal.state = Active \wedge maintenancegoal.state = Active, PEND \rangle$$

and an activation condition that is only satisfied if the maintenance goal is not active. The following example will clarify the behaviour of maintenance and achievement goals in our system.

From the previous example, we have illustrated the rover performing useless actions, and then backtracking in order to recover from violating its maintenance goals. In this example, we illustrate how proactive behaviour eliminates this backtracking and useless actions.

A maintenance goal with proactive behaviour to manage an agent's fuel can be represented as the following.

$$g(refP, fuel = 100, CAP, Maintaining, \pi)$$

where CAP consists of the following condition/action pairs, and where we write $willfail(F)$ for $F \wedge future(\neg F)$.

$$\begin{aligned} &\langle refP.state = \text{Maintaining} \wedge willfail(fuel > distanceToDepot), \text{ACTIVATE} \rangle \\ &\langle refP.state = \text{Active} \wedge fuel == 100, \text{MAINTAIN} \rangle \end{aligned}$$

The first rule states that if the rover believes that in the future, it will be at a point where the fuel is equal or less than the distance to the Depot, this maintenance goal should be activated. This is similar to the reactive behaviour, with the provision that the agent actually anticipate the maintain condition from failing. Note that we also require that the maintain condition is currently true, i.e. that $fuel > distanceToDepot$ to prevent the reactive behaviour from being triggered at the same time as this one.

The second rule states that if the rover was trying to refuel (i.e. this goal's state was active), and the fuel level reached 100%, it should go back to the **Maintaining** state.

The achievement goal requires slight modification, only to indicate that it should not be active when this new maintenance goal is also active. The condition action pairs for this achievement goal are as follows.

$$\begin{aligned} &\langle to6.state = \text{Pending} \wedge ref.state \neq \text{Active} \wedge refP.state \neq \text{Active}, \text{ACTIVATE} \rangle \\ &\langle to6.state = \text{Active} \wedge ref.state = \text{Active}, \text{PEND} \rangle \\ &\langle to6.state = \text{Active} \wedge refP.state = \text{Active}, \text{PEND} \rangle \\ &\langle to6.state = \text{Active} \wedge location = 6, \text{DROP} \rangle \end{aligned}$$

The first rule states that if either maintenance goal is not active and the *to6* goal is pending, it should be activated. The second and third rules state that if either maintenance goal is activated, this achievement goal should move to the pending state to avoid interference. The final rule remains the same, dropping this achievement goal once it has reached its desired location.

As in the previous example, we show how the goal set of the agent evolves as it attempts to achieve the *to6* goal. The initial conditions are the same as before, the rover starting in location 2 with 6 units of fuel. The maintenance goals begin in the **Maintaining** state, while the *to6* goal begins in the **Pending** state. To conserve space, we write *MGR* for $g(ref, \text{Maintaining}, \epsilon)$ and *MPR* for $g(refP, \text{Maintaining}, \epsilon)$. As above, we omit the goal conditions and CAP for each goal as they stay the same throughout execution.

As in the reactive case, we commence by making *to6* active and generating a plan for it.

$$\frac{\frac{\{f(6), MGR, MGP, g(to6, Pending, \pi)\}}{\{f(6), MGR, MGP, g(to6, Active, \pi)\}} \quad 1}{\{f(6), MGR, MGP, g(to6, Active, m36)\}} \quad 10a$$

where we write $m36$ for the plan $m3; m4; m5; m6$.

At this point, we have $willfail(fuel > distanceToDepot)$ being true, as the current fuel level is 6 and the distance to the depot is 2 (so that $fuel > distanceToDepot$) but that $future(fuel \leq distanceToDepot)$ is satisfied. Hence the $to6$ goal goes to the **Pending** state and the proactive maintenance goal MGP is activated. The $to6$ goal is then moved back to the **Pending** state while the maintenance goal is active, which results in the tank being filled.

$$\frac{\frac{\frac{\{f(6), MGR, g(refP, Maintaining, \epsilon), g(to6, Active, m36)\}}{\{f(6), MGR, g(refP, Active, \epsilon), g(to6, Active, m36)\}} \quad 5}{\{f(6), MGR, g(refP, Active, \epsilon), g(to6, Pending, m36)\}} \quad 3}{\{f(6), MGR, g(refP, Active, m1; m0; refuel), g(to6, Pending, m36)\}} \quad 10a}{\{f(100), MGR, g(refP, Active, \epsilon), g(to6, Pending, m36)\}} \quad 11^3$$

Once the tank is filled, the maintain condition is restored, and the maintenance goal goes back to the **Maintaining** state. The $to6$ goal is then resumed, and having found the original plan fails, it finds another plan, which succeeds and so the goal is dropped.

$$\frac{\frac{\{f(100), MGR, g(refP, Active, \epsilon), g(to6, Pending, m36)\}}{\{f(100), MGR, MGP, g(to6, Pending, m36)\}} \quad 4}{\frac{\frac{\{f(100), MGR, MGP, g(to6, Active, m36)\}}{\{f(100), MGR, MGP, g(to6, Active, \epsilon)\}} \quad 1}{\{f(100), MGR, MGP, g(to6, Active, m16)\}} \quad 12}{\frac{\{f(100), MGR, MGP, g(to6, Active, m16)\}}{\{f(94), MGR, MGP, g(to6, Active, \epsilon)\}} \quad 10a}{\{f(94), MGR, MGP\}} \quad 11^6 \quad 7$$

where we write $m16$ for the plan $m1; m2; m3; m4; m5; m6$.

Summary

Note that if we do not have any achievement goals which are pending, then the agent will take no action. For example, given the final state above, i.e. both MGR and MGP in the **Maintaining** state, and with no other goals present, the agent will take no action. This is to be distinguished from the case when the agent has 30 units of fuel, is at location 6 and the goal of moving to location 96 enters the **Pending** state. This goal will be activated, only to

trigger *MGP*, which means the achievement goal goes back to the **Pending** state, and the maintenance goal is activated, which results in the rover moving to location 0 and filling up with fuel. It will then re-attempt the location 96 goal, which is made active. Immediately the *MGP* goal is activated, which causes the location 96 goal to pend. As the goal condition for the *MGP* is satisfied, it returns to the **Maintaining** state. This means that the location 96 goal is made active, and the cycle continues.

While the agent takes no action, it is clearly not desirable to perpetually activate this goal. A solution to this problem may be to use a more sophisticated **future** function, which takes into account the effects of preventative goals. In other words, we do not just test if the location 96 goal will violate the maintain condition, but we also test whether the location 96 goal will still violate the maintain condition after the success of the preventative goal. If both violations occur, as in the example above, the achievement goal should be dropped rather than pended.

There is potential for extended work concerning the **future** function and the formal semantics presented in this chapter. Such work could include developing formal proofs for several features discussed earlier in this these. This includes proving that the reactive behaviour of the maintenance goal is not required (that is, is never pursued) if proactive behaviour is present *and* the **future** function is accurate. Other potential proofs include also showing that both the recovery and preventative goals for a single maintenance goal will never be pursued at the same time.

Ultimately, the accuracy of the **future** function will determine the performance of an agent. In the following chapter, we will address these issues by determining the effectiveness of proactive behaviour for maintenance goals in realistic agent environments, that is, environments which are susceptible to noise and inaccurate data.

Chapter 5

Experimental Results

This chapter serves to provide empirical evidence towards the benefits of proactive maintenance goals. We begin with an explanation of our Mars rover simulator and implementation. We then detail several experiments and provide results that compare aspects of proactive maintenance goals. We conclude with a discussion of these results.

5.1 Experimental Overview

The experiment consists of a simulated Mars rover, based on the description provided in Section 3.1.1. The use of a Mars rover in experiments has been employed in the agent community for many years (for example, Steels [1990]; Thangarajah et al. [2002b]; Meneguzzi and Luck [2007]). Variations and scenarios similar to the Mars rover experiment also exist, for example, the carrier agent example by Hindriks and van Riemsdijk [2007]. Although the specific details may differ between experiments and scenarios, most variants involve autonomous rovers tasked with goals to achieve with limited resources.

The objective of these experiments is to examine the behaviour of maintenance goals, both reactive and proactive, in a variety of settings. To do so, the simulated rover will be given a list of locations to visit. These locations must be visited in-order. The objective of these experiments is not to determine how well an agent can find or optimise a particular route from the locations presented – such a task is akin to solving the well known intractable travelling salesman problem (see Schrijver [2005] for a comprehensive exploration concerning this problem). Instead, we will observe and measure the behaviour and performance of the maintenance goals in managing its fuel level. As described in the following section, the fuel used in each experiment will reflect the efficiency of the agent and the performance of the

maintenance goal employed.

5.1.1 Experimental Setup

In this section, we describe the parameters of the experiment.

The experiment consists of a simulated Mars rover that moves about an environment. The environment represents a planar surface, and we identify locations in this environment via their 2 dimensional Cartesian co-ordinate. For example, Figure 5.1 illustrates several locations in this environment.

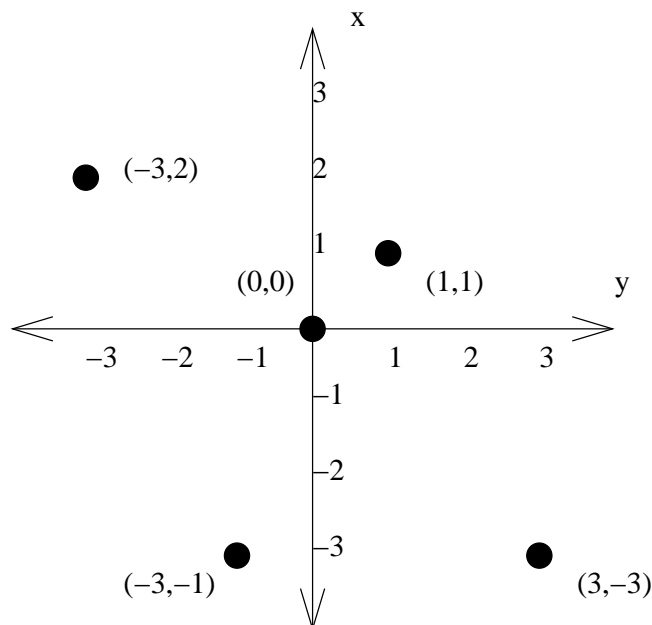


Figure 5.1: Example locations

All locations are within a 40 unit radius from the centre of the map (0,0), where the depot is located. The depot can be used by the Mars rover to refuel its fuel tank to maximum capacity.

The Mars rover can explore this environment by moving in any direction, 1 unit at a time, consuming some fuel in the process. For simplicity, we assume that this is a linear relationship, such that for each unit moved, 1 unit of fuel is consumed. For simplification, there is no cost for turning or breaking, and the rover moves at a fixed speed.

In this experiment, there is always a single depot, which is located at position (0,0). The rover always begins each experiment with a full tank, and starts at the depot. The rover

is able to refill its fuel tank by moving to the depot and performing the refuel action. We assume that the depot has an unlimited supply of fuel for this experiment.

In this experiment, the rover has the task of visiting several locations. The number of locations to visit varies for each experiment. In this case, the number of locations vary between 10,100,1000 and 10000 randomly generated locations. In our experimental results however, we only provide the results for 10 and 10000 locations – these are adequate for illustrating our findings and displaying trends in the data from all cases.

A location is a single point in the environment, represented as a (x,y) pair. The straight line distance between any location to visit and the depot is less than 40 units. This is to ensure that it is possible for the rover to visit all locations after refuelling with the smallest tank (100 units). As the maximum distance to a location is 40, a trip to the most remote location from the depot and back will be 80 units, which is possible with the 100 unit fuel tank, with a 20 unit buffer.

To present a more realistic environment, we also vary the accuracy of the agent’s estimation. The agent estimates the distance between various locations, its current location, and the depot at various times. There are two parameters to defining the error rate, the upper and lower bounds, which are given in percentages of the correct value.

An error rate of plus or minus 20% has an upper bound of 120%, and a lower bound of 80%. Therefore, if an estimation is made on a distance that is 10 units, the estimation will return a value between 9 and 11 inclusive.

Fixed error rates can also be used by fixing the upper and lower bounds to be equal. For example, an error rate with an upper bound of 150% and a lower bound of 150% will always overestimate the correct distance by half – 20 unit distance will have an estimate of 30 units. A similar approach can be used to always underestimate.

Each new location presented to the rover is represented by an achievement goal, **MoveTo**, which moves the rover to a specified location. An appropriate plan for such a goal is to generate a sequence of unit steps in a straight line between the rovers current location and the location it intends to reach. The rover processes only a single goal each time, ensuring that the locations are visited in the presented order.

The agent also has two maintenance goals, one with reactive behaviour and the other with proactive behaviour, which are both initially adopted (maintaining). The reactive maintenance goal becomes active when its fuel level is less than or equal to the amount of fuel remaining in its fuel tank. The proactive maintenance goal becomes active when it predicts that, based on the currently adopted goal and appropriate plan, the fuel level will be less

than or equal to the remaining fuel **and** the reactive maintenance goal is not currently active. In both cases, an appropriate plan for either of these goals is to move to the depot (0,0) and refuel. Success is denoted once the fuel level returns to maximum capacity (either 100 or 200).

5.1.2 Definitions and Terminology

In this section, we will use particular terminology to describe the outcomes of each trial, and what we measure in each trial.

Measurement of each trial

Goal-directed distance This is the distance travelled when the rover moves from the location it adopted the goal to the goal location, given that it does so uninterrupted. Essentially, this is the shortest distance between the location it adopted the goal, and the goal location.

Backtrack distance This distance represents the distance the rover moves towards the refuelling depot.

Waste distance This distance represents the additional distance the rover travelled that could have been avoided by moving directly to the depot.

It is important to note that we consider both goal-directed and backtrack distances essential to normal operation of the rover. The objective is to reduce or eliminate waste distance when possible.

Figure 5.2 illustrates these components.

In the graphs presented in this chapter, goal-directed distances are represented by white colour, backtrack distance represented by grey, and waste represented as black.

Ideally, an agent should use as little fuel as possible in achieving its goals. To this end, the smaller the *waste*, the better the performance of the agent. Minimising the goal-directed and back-tracking distances, while possible, is a much harder problem.

Outcomes of trials

In addition to these terms that describe components of each trial, the following terms will be used to describe the results of each trial.

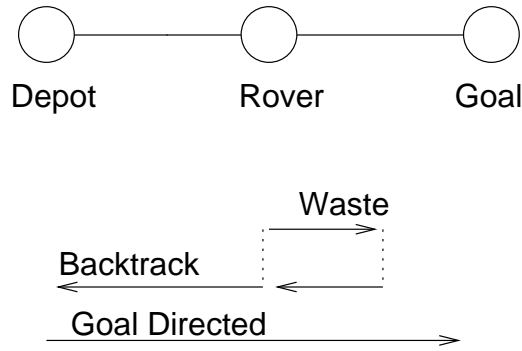


Figure 5.2: Categorisation of movement types

Complete If a trial is described as *complete*, it indicates that the agent visited all the locations in the correct order, and did not run out of fuel at any time.

Stranded A *stranded* trial indicates that the agent failed to visit all the locations as it ran out of fuel sometime during its journey. We consider this a bad result, as it indicates that in a real situation, the rover would be stranded without fuel.

This outcome is only possible if the agent under estimates how much fuel it requires to return to the depot to refuel. If it requires 10 units of fuel to return, but the agent believes only 5 units is required, it will only take action when 5 units remains – this is too late at this time. This outcome is only possible when error is introduced into the simulation.

Halted A trial denoted as *halted* indicates an occasion where the agent believes that it is impossible to achieve a goal, and so does not attempt to pursue it. It therefore stops processing all goal and the simulation stops. This is quite different than a *stranded* trial, as the agent is not stranded. In this case, it is performing a rational action, as its beliefs dictate that the goal is impossible to achieve.

Looping A trial that is described as *loops* indicates a trial where the rover is caught in a loop. In attempting to move to some location, a maintenance goal is triggered and so the agent moves to the depot to refuel. It then attempts to move to the original location, but again, the maintenance goal is triggered, and the cycle repeats. The agent cannot progress as it is trapped by its maintenance goal. This outcome is possible in a situation without errors when using a naïve approach to reactive maintenance goals. If a goal is given that is impossible to reach, the agent will attempt to move towards

the target, but require refuelling when its fuel tank is half full. It returns to the depot to refuel, only to re-attempt the goal ad infinitum.

This is implemented in practice by stopping any simulation that takes over 1,000,000 steps to complete. This is much larger than any possible route that does complete.

We now summarise the experimental setup.

Each run of the experiment consist of a rover moving around a particular map. Each map consists of a fixed number of locations, ranging from 10 to 10000 locations. Each location is randomly generated, but its distance from (0,0) is always less than or equal to 40 units. The rover has a varying size fuel tank, either 100 or 200 units in capacity. As the rover moves, it consumes fuel, at the rate of 1 unit of fuel consumed for every 1 unit of distance travelled.

We measure the goal-directed, backtrack and waste distances for each trial, as well as the overall outcome which is one of either complete, stranded, halted or looping. At times, the rover will need to perform estimations – these estimations may be incorrect, influenced by the error rate of the particular experiment. We define the error rates for the reactive and proactive estimations separately, as they may use different algorithms in practice.

5.2 Hypothesis

The purpose of these experiments is to verify the following statements.

- In an error free environment, if the proactive maintenance goal is present, the reactive maintenance goal is never activated.
- In an error prone environment, as the error rate increases, the reactive maintenance goal is activated more often.
- In an error prone environment, the performance of proactive maintenance goal degrades gracefully.

5.3 Results

We will discuss our findings in several parts. We will look at how maintenance goals behave in environments with varying degrees of errors, beginning firstly with error free environments, then to environments that consistently over and under estimate the true distance to goals, before concluding with environments that are capable of *both*, over and under estimating.

5.3.1 Maintenance goals in error-free environments

In this first situation (Figures 5.3 through 5.6), the estimation of the distance to be travelled by the rover is always correct. We compare how reactive maintenance goals and proactive maintenance goals behave in this environment for three different fuel tank capacities, and several different sized number of locations to visit.

As expected, the cases with proactive maintenance goals out performs the cases where only reactive maintenance goals are used. In all cases, all goals were achieved successfully. As the capacity of the fuel tank increased, the proportion of waste and backtracking distance compared with the overall distance travelled decreased.

On average, waste accounted for approximately 25 percent of the total distance travelled in the reactive case, and zero when proactive maintenance goals were also used, with 100 units of fuel. The total distance travelled when using proactive maintenance goals compared with not using them is approximately 75%. Therefore, using proactive maintenance goals saved (on average) 25% fuel.

5.3.2 Varying errors in Maintenance goals

In the previous experiment, the environment was error free, and therefore, estimations were always correct. In the real world, however, it is likely that correct estimation cannot be relied upon. In this experiment, we randomly assign an error rate each time the rover needs to perform a distance check. The range of the error is limited to plus or minus 10, 20, 30, 40, 50, 60, 70, 80, 90 and 100%. For example, with an error rate of plus or minus 20%, and a true distance of 10 units, the estimated distance could be any value between 8 and 12 units. The purpose of this experiment is to determine how well maintenance goals behave in more realistic environments.

Figures 5.7 through ?? illustrate the outcomes of this experiment. Most noticeable is that as the error rates increase, we observe that the rover is unable to visit all the goals and complete a trial. This is due to the rover becoming *stranded* at some point. Similar results are present for all map sizes, with the exception that this failure occurs for all error rates, other than 0%.

When using proactive maintenance goals, some successful trials were completed with error rates up to $\pm 20\%$. However, this occurred only in the smallest map size. Reactive maintenance goals failed in the presence of errors.

It appears that in the cases where there were errors and small map size, the rover was

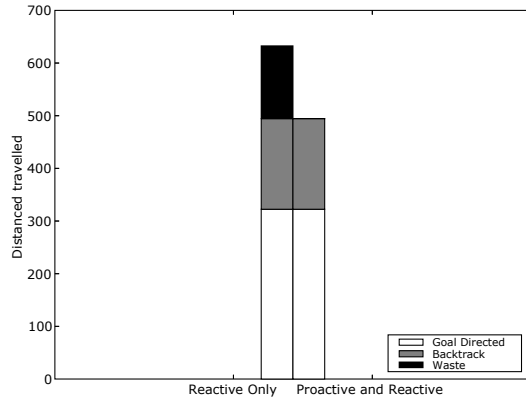


Figure 5.3: 10 locations, fuel tank of 100

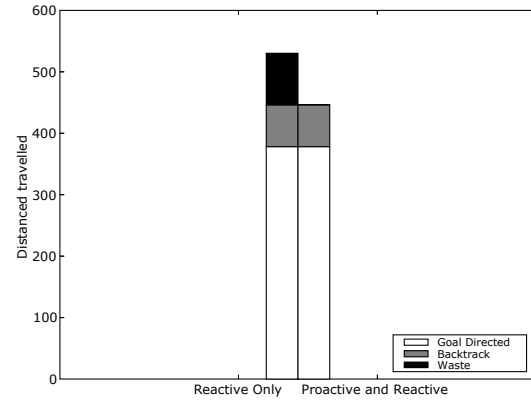


Figure 5.4: 10 locations, fuel tank of 200

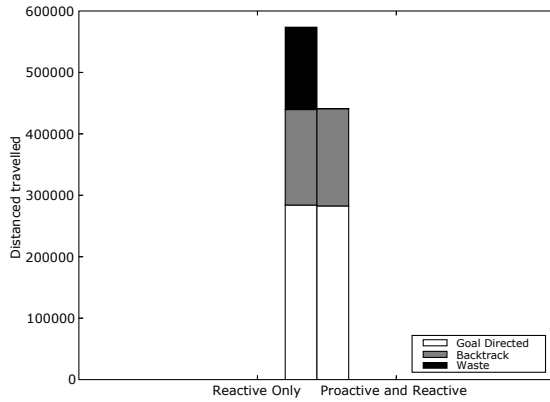


Figure 5.5: 10000 locations, fuel tank of 100

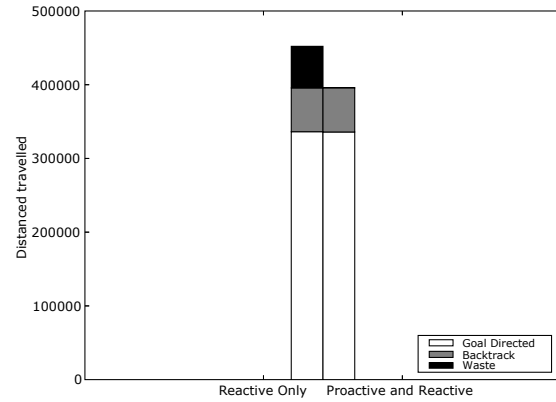


Figure 5.6: 10000 locations, fuel tank of 200

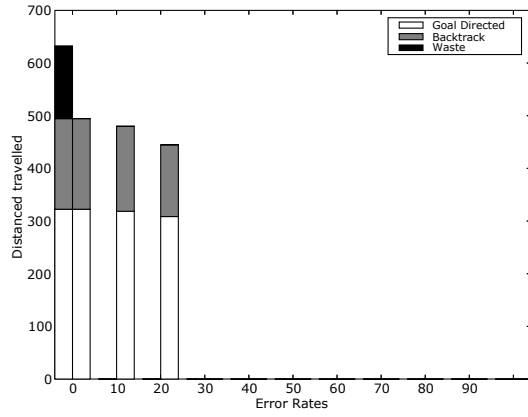


Figure 5.7: 10 locations, fuel tank of 100

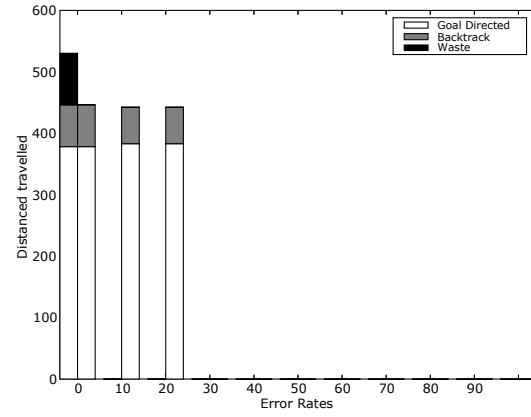


Figure 5.8: 10 locations, fuel tank of 200

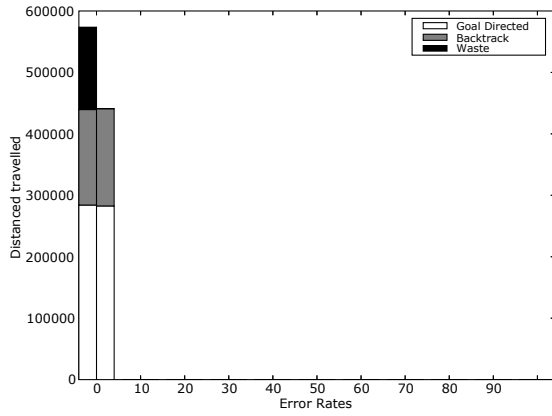


Figure 5.9: 10000 locations, fuel tank of 100

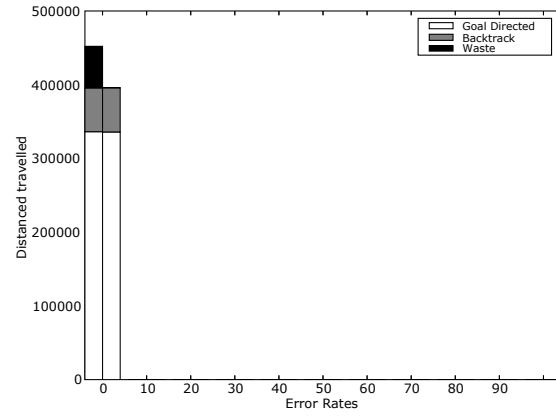


Figure 5.10: 10000 locations, fuel tank of 200

‘lucky’ and its estimations were generally safe. As the map size increased, and therefore more estimations performed, it is more likely that a poor estimation is generated which leads the rover to becoming stranded.

Even doubling the maximum capacity of the fuel tank did little to assist the success rate of the rover in the prescience of errors. As expected, it reduced the amount of fuel used when successful.

In the next experiments, we will aim to determine the effects overestimation and underestimation have on maintenance goals separately.

5.3.3 Overestimation in Maintenance goals

In this experiment, the rover overestimates the distance to be travelled. For example, if the true distance to travel is 10 units, and there is an error rate of 120%, the rover believes it needs to travel 12 units. Figures 5.11 through 5.14 summarise the results of this experiment.

In this experiment, we found that as error rates increased, successful completion of all goals decreased. This is most apparent in the case of a limited fuel tank. The reason for this is that when overestimating to a high or moderate degree means that the agent believes that some locations are too far to visit and return to the depot, even with a fully stocked fuel tank. Therefore, the agent *halts* all future goals. In the case where only reactive maintenance goals are used, the result of these failed attempts often results in looping behaviour. The only possible outcomes are for the rover to visit all locations (*complete*), *loop* in the case of only reactive maintenance goals, or *halt* in the case when also using proactive maintenance goals.

In the case of the 200 unit fuel tank, all attempts were successful, even with an error rate of 200%. This is because the maximum distance a goal can be from the depot is 40 units – even with 200% error, the rover will believe it to be 80 units from the depot, so a trip there and back to the depot is under the 200 unit fuel cap.

Furthermore, it was generally the case that not using proactive maintenance goals allowed the rover to successfully visit all the goals at higher error rates than when using proactive maintenance goals. After the error rate of 130%, cases using the proactive maintenance goal began to fail, where as the reactive maintenance goal continued to have 100% completion until 160%.

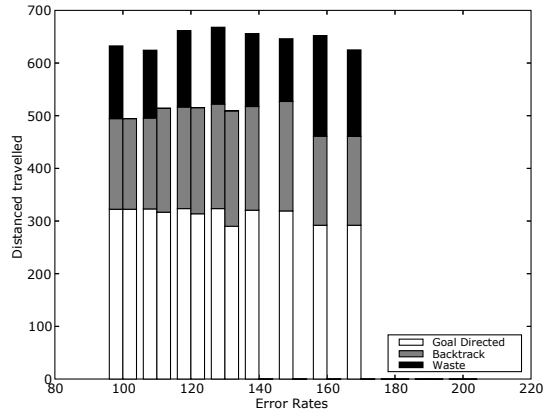


Figure 5.11: 10 locations, fuel tank of 100

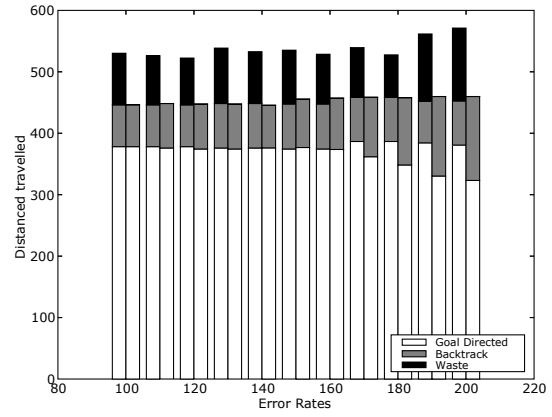


Figure 5.12: 10 locations, fuel tank of 200

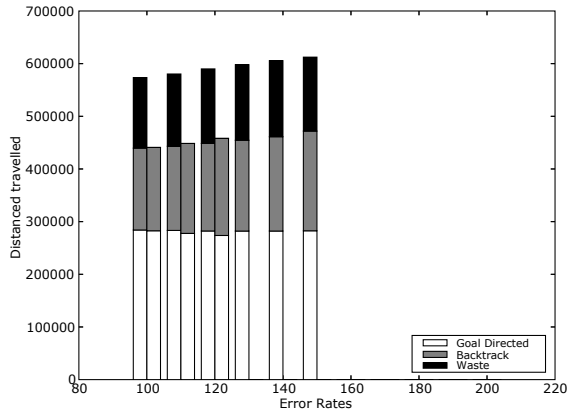


Figure 5.13: 10000 locations, fuel tank of 100

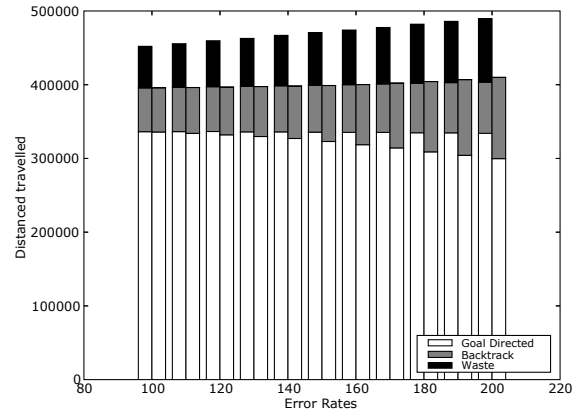


Figure 5.14: 10000 locations, fuel tank of 200

5.3.4 Underestimation in Maintenance goals

In this experiment, the rover continually underestimates distances. For example, if the true distance was 10 units and the error rate was 80%, the rover would expect that it needs to travel 8 units.

In this experiment, underestimation severely limited the success rate of the rover. Once the number of locations to visit exceeded 10, it was rare that all goals were achieved.

The results of this experiment are listed in Figures 5.15 through 5.18. In these situations, using proactive maintenance goals in conjunction with reactive maintenance goals fared slightly better than using reactive maintenance goals alone. In these cases, reactive maintenance goals become *stranded* as soon as underestimation was introduced, while also using proactive maintenance goals allowed small amounts (up to 80% accuracy) of underestimation to occur and still have some attempts *complete* successfully.

Once the number of goals increase beyond 10 locations however, only reactive and reactive with proactive maintenance goals have the same results, failing due to becoming *stranded*. Due to the underestimation, the maintenance goals are triggered at a later stage than required, therefore the rover's fuel is often inadequate to return to the depot. For some of the cases, especially in the cases with a small number of locations, the rover may be fortunate and not need to refuel to visit all locations. As the number of goals increases however, it becomes more likely that the refuelling will be required, but the rover's fuel inadequate.

Increasing the size of the fuel tank only aided slightly in increasing the number of attempts successfully completed, with a map size of 10. Even with a 200 unit fuel tank, when underestimation error was present, all goals could not be *completed* in any attempt, instead, the rover fails its task by becoming *stranded*.

Summary

We will summarise our experimental evaluation now. Firstly, experiment 1 demonstrated that in ideal settings, proactive maintenance goals lead to more efficient behaviour than using reactive maintenance goals alone. In this ideal setting, the reactive maintenance goal was never employed (that is, there was no waste) when proactive maintenance goals were present. As resource availability increases, the proportion of waste decreases.

When estimating, positive and negative error rates greatly effect the performance and success rate of the rover experiments. In almost all cases, the rover became stranded. Further experiments were required to determine the cause of these failures.

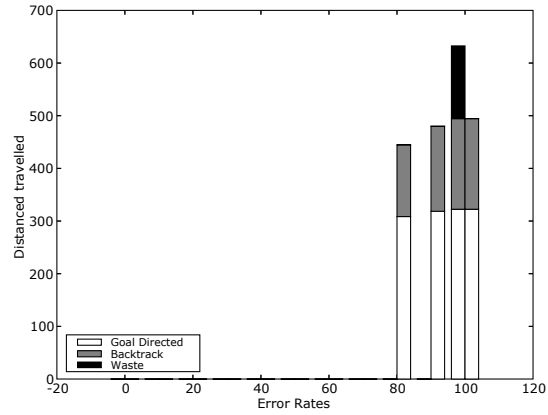


Figure 5.15: 10 locations, fuel tank of 100

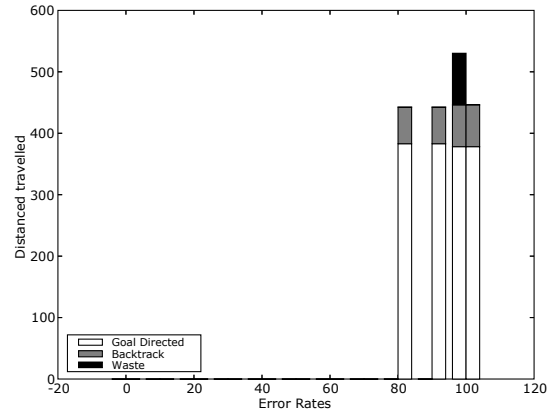


Figure 5.16: 10 locations, fuel tank of 200

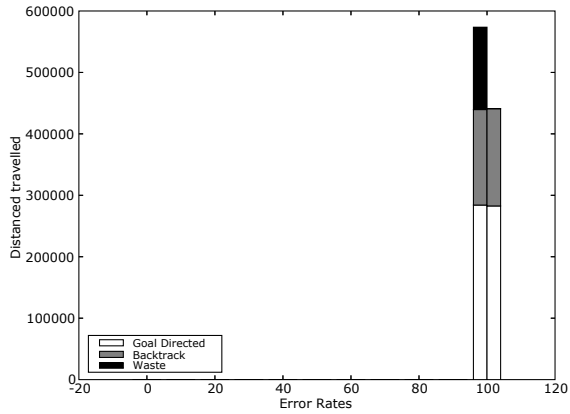


Figure 5.17: 10000 locations, fuel tank of 100

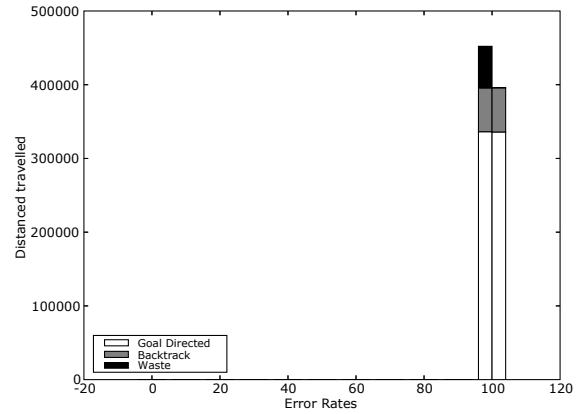


Figure 5.18: 10000 locations, fuel tank of 200

When overestimating, problems can occur in both reactive and proactive maintenance goals. This is especially apparent in the proactive case. The agent can't act 'over-zealous' and not attempt goals that it believes it cannot do (which is rational), as its beliefs state that the goal is impossible. When goals are close to the limits of the agent, overestimation can cause the agent to believe these goals are impossible, while they actually can be achieved. Instead, the agent halts action. When adequate resources are present however (as in the case of the 200 capacity fuel tank), no such problems arise, and the increase in error rate only causes slight increase in the consumption of resources.

Underestimating can lead to problems for both reactive and proactive maintenance goals. In the case of reactive maintenance goals, looping can occur, leading to a huge consumption of resources with little benefit. These problems can be addressed by the inclusion of some additional reasoning in the deliberation portion of the agent, to avoid repeated attempts at goals where it is impossible.

Chapter 6

Conclusion

The use of intelligent agents is increasing, especially in cases where requirements include timely response, goal-directed behaviour, in environments that have the potential to change rapidly over time. Goal based agents, such as those that follow the BDI paradigm, are particularly suited to these tasks.

Achievement goals are the most common form of goal found in agent systems, driving agents to perform actions in order to accomplish these goals. Maintenance goals are also common, but rather than realising some goal, cause an agent to perform actions in order to keep some state true.

Current implementations of the agent paradigm, such as Jadex, JACK and Jam, have support for maintenance goals. In these frameworks however, maintenance goals are utilised in a reactive manner.

Reactive maintenance goals have been shown to be limited in their ability, as they generally act as triggers for plans or achievement goals, which is similar to the manner in which achievement goals are utilised. These maintenance goals have no influence over the agent's behaviour until an associated maintenance condition is no longer met, which causes the agent to attempt to repair the condition.

In contrast to this, we have developed the notion of proactive maintenance goals, which we have shown to be useful in a variety of scenarios. Maintenance goals with proactive behaviour influence the agent's behaviour continually, aiming to ensure that the maintenance condition is *never* violated. If the agent can anticipate that failure is imminent, based on the future plans of the agent, it performs actions that aim to prevent failure occurring. In cases where no action is available that avoids the failure of maintenance goals, the original actions should

not be pursued. This is an example of a constraint – as there is no appropriate recovery or preventative action is possible, therefore the only satisfactory solution is to avoid actions that lead to maintenance condition failure.

After the analysis of the behaviour of maintenance goals in several case studies, we developed a representation of maintenance goals that captured both reactive and proactive behaviours. Algorithms for reasoning about these maintenance goals were developed in Chapter 3, and were then formalised in Chapter 4.

Our formalism was used to illustrate and prove various ideals discussed by analysing the case studies. It further developed an agent language by van Riemsdijk et al. [2008] to incorporate maintenance goals with stronger notions of proactivity.

In Chapter 5, we took an experimental approach to analysing the effectiveness of maintenance goals. Several experiments that were conducted illustrated that using proactive maintenance goals in addition to reactive maintenance goals outperformed reactive maintenance goals alone. One important variable was the error rate in the agent’s perception of the environment. This was altered to reflect occasions when the agent underestimated the true resource requirements, as well as when it overestimated these requirements.

Our experimental findings show that when an agent checks its maintenance conditions, the process employed with proactive maintenance goals should not underestimate, as this has the potential to lead to an agent abandoning goals as it believes them to be impossible. This has particular significance as the method of determining if maintenance conditions will be violated due to the agent’s planned actions is based on the agent’s beliefs, which *can* be inaccurate. Our findings showed that in *all* cases, the addition of proactive maintenance goals, even when error prone, reduced the amount of resources consumed, as well as avoiding some situations where the rover became stranded. In these cases, the agent intelligently refuses to commence achieving a goal which it believes impossible to complete.

In the case of reactive maintenance goals, we determine that these goals should not overestimate, as this can cause them to run low on resources prematurely. As demonstrated in the experiments, in some cases, the result of this can be severe, such as the rover becoming stranded. In this case, the rover has run out of fuel away from the depot, and has no means to recovery.

Future work

While proactive maintenance goals have been identified and introduced in this thesis, much work remains as to how maintenance goals will be utilised in agent systems in the future.

The **future** construct described in this thesis utilises prediction mechanisms that are simplistic in nature. In this thesis, we have focused on the use of resource summaries as one such mechanism for determining the state of the world after the execution of an agent's plan(s). Resource summaries are not the only approach however, and we have discussed alternatives such as user supplied methods. We believe that the experiments we performed using the resource summary heuristic can act as a guide for how alternate heuristics may perform when inaccurate. Suggestions for alternate implementations of **future** could include planning-based approaches, historical-based approaches, or perhaps neural networks. Their suitability for this task could be investigated in the future.

One idea is the concept of *pruning*. At the moment, when an achievement goal cause conflict with a maintenance condition, the achievement goal is suspended until the maintenance goal's preventative (or recovery) goal is performed. An alternate approach may be to determine which particular portions of the goal (such as a sub-goal or plan) is causing the conflict, and finding an alternative to this subset. This is of particular importance for solutions which use a planning-approach to plan generation and selection. Refinement of a goal, as described by Hindriks and van Riemsdijk [2007] is another possibility for integration with our approach. If a goal of several parts is causing conflict, it may be possible to remove or weaken portions of the goal so that it no longer causes conflict.

We consider the Mars rover case study as an example. It may have a combined goal of visiting 3 locations near each other, but the depot has no fuel available, and there is limited fuel in the rover. Rather than avoid completing all the goals, it may prune this goal to visit only a single location instead, which it can do with the remaining fuel that it has. The process for accomplishing such results, as well as determining when this approach is warranted, should be investigated in the future. Aspects that need to be addressed include how it determines which goals can be pruned away and which goals must be retained, as well as mechanisms for ensuring that compound goals cannot be split or separated.

In addition to pruning, an alternative is to consider how best to integrate maintenance goals into the *means-end-reasoning* process. In this case, a planner will most likely be used, as an alternative to selecting plans from a library. Hindriks and van Riemsdijk [2007] provided one such approach, by means of pruning achievement goals that caused conflict with

maintenance goals. This was achieved by looking at all the possible plans steps, which is likely to be infeasible in practice. One approach that overcomes this may be to introduce a notion of a planning horizon – a limit to how many plan steps deep any search will reach. A reasonable limit would be to search the limits of all plan steps in the goals that have already been adopted. These are the goals that the agent is currently pursuing, with a certain selected plan. Plans may cause subgoals to be adopted however, and further means-end-reasoning could be performed once these goals are adopted. Therefore, the agent only needs to search over what it already intends to perform, rather than over all possible actions. How such a mechanism works in practice is an item of future work.

The specific aims of this thesis were to research and develop a more sophisticated notion of maintenance goals for intelligent agents. To this end, the notion of treating maintenance goals *proactively* has been introduced, including requirements for representation and mechanism for their use, as well as operational semantics that support this representation. To illustrate the gains of proactive maintenance goals over reactive maintenance goals, a number of experiments were conducted that illustrated that in most cases, proactive maintenance goals outperformed reactive maintenance goals.

Bibliography

- P. E. Agre and D. Chapman. Pengi: An implementation of a theory of activity. *Proceedings of the Sixth National Conference on Artificial Intelligence*, 278, 1987.
- C. Baral and T. Eiter. A polynomial-time algorithm for constructing k-maintainable policies. In D. Dubois, C. A. Welty, and M.-A. Williams, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004)*, Whistler, Canada, June 2-5, 2004, pages 720–730. AAAI Press, 2004.
- C. Baral, T. Eiter, M. Bjärelund, and M. Nakamura. Maintenance goals of agents in a dynamic environment: Formulation and policy construction. *Artificial Intelligence*, 172 (12-13):1429–1469, 2008.
- J. Bell and Z. Huang. Dynamic Goal Hierarchies. In *Proceedings from the Workshop on Intelligent Agent Systems, Theoretical and Practical Issues*, pages 88–103. Springer-Verlag, 1997.
- F. Bellifemine, A. Poggi, and G. Rimassa. JADE - a FIPA-compliant agent framework. In *Proceedings of the Practical Applications of Intelligent Agents*, volume 99, pages 97–108, London, UK, 1999. The Practical Application Company Ltd.
- R. H. Bordini and J. F. Hübner. BDI agent programming in agentspeak using jason (tutorial paper). In F. Toni and P. Torroni, editors, *CLIMA VI*, volume 3900 of *Lecture Notes in Computer Science*, pages 143–164. Springer, 2005.
- M. E. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.
- L. Braubach, A. Pokahr, W. Lamersdorf, and D. Moldt. Goal representation for BDI agent systems. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *Sec-*

- ond *International Workshop on Programming Multiagent Systems: Languages and Tools*, pages 9–20, 2004.
- R. Brooks. Integrated systems based on behaviors. *ACM SIGART Bulletin*, 2(4):46–50, 1991a.
- R. A. Brooks. Intelligence without reason. In R. Myopoulos, John; Reiter, editor, *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 569–595, Sydney, Australia, 1991b. Morgan Kaufmann.
- R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1-3):139–159, 1991c.
- R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. iGRAIL/KAOS: An environment for goal-driven requirements engineering. In *ICSE '97: Proceedings of the 19th International Conference on Software Engineering*, pages 612–613, New York, NY, USA, 1997. ACM.
- M. Dastani, F. Dignum, and J. Meyer. 3APL: A Programming Language for Cognitive Agents. *ERCIM News, European Research*, (53):28–29, 2000.
- M. Dastani, M. B. van Riemsdijk, F. Dignum, M. Birna, R. F. Dignum, and J. Jules Ch. Meyer. A programming language for cognitive agents goal directed 3APL. In *Proceedings of the First International Workshop on Programming Multiagent Systems 2003*, pages 111–130. Springer, 2003.
- M. D’Inverno, M. Luck, M. Georgeff, D. Kinny, and M. Wooldridge. The dMARS Architecture: A Specification of the Distributed Multi-Agent Reasoning System. volume 9, pages 5–53. Springer, 2004.
- R. Evans. Varieties of learning. In *AI Game Programming Wisdom*, pages 567–578. Charles River Media, 2002.
- I. A. Ferguson. *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents*. PhD thesis, Univeristy of Cambridge, UK, 1992a.
- I. A. Ferguson. Towards an architecture for adaptive, rational, mobile agents. *Decentralized AI*, 3:249–262, 1992b.

- R. Fikes and N. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.
- S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *ECAI '96: Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages*, pages 21–36, London, UK, 1997. Springer-Verlag.
- M. Georgeff, B. Pell, M. Pollack, M. Tambe, and M. Wooldridge. The belief-desire-intention model of agency. In J. Müller, M. P. Singh, and A. S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 1–10, Heidelberg, Germany, 1999. Springer-Verlag.
- M. P. Georgeff and F. F. Ingrand. Decision-making in an embedded reasoning system. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, volume 2, pages 972–978. Morgan Kaufmann, 1989.
- K. V. Hindriks and M. B. van Riemsdijk. Satisfying maintenance goals. In M. Baldoni, T. C. Son, M. B. van Riemsdijk, and M. Winikoff, editors, *Declarative Agent Languages and Technologies V, 5th International Workshop, DALT 2007, Honolulu, HI, USA, May 14, 2007, Revised Selected and Invited Papers*, volume 4897 of *Lecture Notes in Computer Science*, pages 86–103. Springer, 2007.
- K. V. Hindriks and M. B. van Riemsdijk. Using temporal logic to integrate goals and qualitative preferences into agent programming. In M. Baldoni, T. C. Son, M. B. van Riemsdijk, and M. Winikoff, editors, *Declarative Agent Languages and Technologies VI: 6th International Workshop*, volume 5397, pages 215–232. Springer-Verlag, 2008.
- K. V. Hindriks, F. S. De Boer, W. Van Der Hoek, and J.-J. Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.
- M. J. Huber. Jam: a BDI-theoretic mobile agent architecture. In *AGENTS '99: Proceedings of the third annual conference on Autonomous Agents*, pages 236–243, New York, NY, USA, 1999. ACM.
- J. F. Hübner, R. H. Bordini, and M. Wooldridge. Plan patterns for declarative goals in agentspeak. In *AAMAS '06: Proceedings of the Fifth International Joint Conference on*

- Autonomous Agents and Multiagent Systems*, pages 1291–1293, New York, NY, USA, 2006. ACM.
- G. A. Kaminka, A. Yakir, D. Erusalimchik, and N. Cohen-Nov. Towards collaborative task and team maintenance. In *AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–8, New York, NY, USA, 2007. ACM.
- D. N. Kinny. Measuring the effectiveness of situated agents. Technical report, Australian AI Institute, 1990.
- J. Lee, M. J. Huber, E. H. Durfee, and P. G. Kenny. UM-PRS: An implementation of the procedural reasoning system for multirobot applications. In *Conference on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS)*, volume 94, pages 842–849, 1994.
- M. Ljungberg and A. Lucas. The OASIS air-traffic management system. In *Proceedings of the Second Pacific Rim International Conference on Artificial Intelligence (PRICAI '92)*, pages 236–243, Seoul, Korea, 1992.
- P. Maes. Modeling Adaptive Autonomous Agents. *Artificial Life*, 1(1):135–162, 1994.
- P. Maes. The dynamics of action selection. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 2:991–997, 1989.
- P. Maes. Situated agents can have goals. *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, pages 49–70, 1990.
- P. Maes. The agent network architecture (ANA). *ACM SIGART Bulletin*, 2(4):115–120, 1991.
- F. R. Meneguzzi and M. Luck. Motivations as an abstraction of meta-level reasoning. In H.-D. Burkhard, G. Lindemann, R. Verbrugge, and L. Z. Varga, editors, *CEEMAS 2007: Proceedings of the 5th international Central and Eastern European conference on Multi-Agent Systems and Applications V*, volume 4696 of *Lecture Notes in Computer Science*, pages 204–214. Springer, 2007.
- J. Muller. A Conceptual Model for Agent Interaction. *Proceedings of the 2nd International Working Conference on Cooperative Knowledge Based Systems (CKBS-94)*, Keele University: Dake Centre, pages 213–234, 1994.

- J. Muller and M. Pischel. Modelling interacting agents in dynamic environments. *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI-94)*, pages 709–713, 1994.
- J. Muller, M. Pischel, and M. Thiel. A pragmatic approach to modelling autonomous interacting systems: Preliminary report. *Intelligent Agents: Theories, Architectures, and Languages*, 890, 1994.
- N. Muscettola, P. Nayak, B. Pell, and B. Williams. Remote Agent: to boldly go where no AI system has gone before. *Artificial Intelligence: 40 Years Later*, 103(1-2):5–47, 1998.
- M. Nakamura, C. Baral, and M. Bjärelund. Maintainability: A weaker stabilizability like notion for high level control. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 62–67. AAAI Press / The MIT Press, 2000.
- P. D. O’Brien and R. C. Nicol. FIPA—Towards a Standard for Software Agents. *BT Technology Journal*, 16(3):51–59, 1998.
- A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: Implementing a BDI-infrastructure for jade agents. *EXP - in search of innovation (Special Issue on JADE)*, 3(3):76–85, 2003.
- A. Pokahr, L. Braubach, and W. Lamersdorf. A BDI architecture for goal deliberation. In S. K. S. K. M. P. S. Frank Dignum, Virginia Dignum and M. Wooldridge, editors, *AAMAS ’05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 1295–1296, New York, NY, USA, 2005a. ACM.
- A. Pokahr, L. Braubach, and W. Lamersdorf. A goal deliberation strategy for BDI agent systems. In W. L. M. K. M. H. T. Eymann, F. Klg, editor, *Third German conference on Multi-Agent System TEchnologieS (MATES-2005); Springer-Verlag, Berlin Heidelberg New York, pp. 82-94*, pages 82–94. Springer-Verlag, Berlin Heidelberg New York, 2005b.
- A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *MAA-MAW ’96: Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away*, pages 42–55, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

- A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In *Principles of Knowledge Representation and Reasoning. Proceedings of the second International Conference*, pages 473–484, San Mateo, 1991. Morgan Kaufmann.
- A. S. Rao and M. P. Georgeff. BDI-agents: From Theory to Practice. In *Proceedings of the First International Conference on Multiagent Systems (ICMAS'95)*, pages 312–319, San Francisco, 1995. MIT Press.
- S. Rosenschein and L. Kaelbling. The synthesis of digital machines with provable epistemic properties. *Proceedings of the 1986 conference on Theoretical aspects of reasoning about knowledge*, pages 83–98, 1986.
- A. Schrijver. On the history of combinatorial optimization (till 1960). In K. Aardal, G. Nemhauser, and R. Weismantel, editors, *Handbook of Discrete Optimization*, pages 1–68. Elsevier, Amsterdam, 2005.
- L. Steels. Cooperation between distributed agents through self-organization. In Y. Demazeau and J.-P. Müller, editors, *IEEE International Workshop on Intelligent Robots and Systems 1990*, pages 8–14 suppl., New York, NY, USA, 1990. IEEE.
- J. Thangarajah, L. Padgham, and J. Harland. Representation and reasoning for goals in BDI agents. In M. J. Oudshoorn, editor, *Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*, volume 4 of *CRPIT*, pages 259–265, Melbourne, Australia, 2002a. ACS.
- J. Thangarajah, M. Winikoff, L. Padgham, and K. Fischer. Avoiding resource conflicts in intelligent agents. In *Proceedings of the 15th European Conference on Artificial Intelligence 2002 (ECAI 2002)*, pages 18–22. IOS Press, 2002b.
- J. Thangarajah, L. Padgham, and M. Winikoff. Detecting and avoiding interference between goals in intelligent agents. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI2003)*, pages 721–726. Academic Press, 2003a.
- J. Thangarajah, L. Padgham, and M. Winikoff. Detecting and exploiting positive goal interaction in intelligent agents. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 401–408, New York, NY, USA, 2003b. ACM Press.

- B. van Linder, W. van der Hoek, and J.-J. C. Meyer. Formalising motivational attitudes of agents. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *ATAL*, volume 1037 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 1995.
- B. van Riemsdijk, W. van der Hoek, and J.-J. C. Meyer. Agent programming in dribble: From beliefs to goals using plans. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 393–400, New York, NY, USA, 2003. ACM.
- M. B. van Riemsdijk, M. Dastani, and M. Winikoff. Goals in agent systems: A unifying framework. In *AAMAS '08: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 713–720, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.
- M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative procedural goals in intelligent agent systems. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002)*, pages 470–481, 2002.
- M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.