Managing the Concurrent Execution of Goals in Intelligent Agents

A thesis submitted for the degree of
Doctor of Philosophy

John Thangarajah,
B.App.Sc.Computer Science,
B.App.Sc.Computer Science Honours,
School of Computer Science and Information Technology,
Faculty of Applied Science,
Royal Melbourne Institute of Technology,
Melbourne, Victoria, Australia.

May 27, 2005

Declaration

This thesis contains work that has not been submitted previously, in whole or in part, for any other academic award and is solely my original research, except where acknowledged. The work has been carried out since the beginning of my candidature on 01 April 2001.

Preliminary versions of some results or discussions in this thesis have been previously published:

- Thangarajah, J. & Padgham, L. (2004), An Empirical Evaluation of Reasoning about Resource Conflicts in Intelligent Agents, *in* 'Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS 2004)', New York, USA.

- Thangarajah, J., Padgham, L. & Winikoff, M. (2003), Detecting and Exploiting Positive Goal Interaction in Intelligent Agents, *in* 'Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-agent Systems', ACM Press, Melbourne, Australia.

- Thangarajah, J., Padgham, L. & Winikoff, M. (2003), Detecting and Avoiding Interference between Goals in Intelligent Agents, *in* 'Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003)', Acapulco, Mexico.

- Thangarajah, J., Winikoff, M., Padgham, L. & Fischer, K. (2002), Avoiding Resource Conflicts in Intelligent Agents, *in* 'Proceedings of the Fifteenth European Conference on Artificial Intelligence 2002 (ECAI 2002)', Lyon, France.

- Winikoff, M., Padgham, L., Harland, J. & Thangarajah, J. (2002), Declarative and Procedural Goals in Intelligent Agent Systems, *in* 'Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002)', Toulouse, France.

- Thangarajah, J., Padgham, L. & Harland, J. (2002), Representation and Reasoning for Goals in BDI Agents, *in* 'Proceedings of the Twenty-Fifth Australasian Conference on Computer science', Australian Computer Society, Inc., Melbourne, Australia.

John Thangarajah
School of Computer Science and Information Technology
Royal Melbourne Institute of Technology
May 27, 2005

Acknowledgements

I would firstly like to thank my supervisors, Associate Professor Lin Padgham and Dr. Michael Winikoff, for the support and guidance they have given me throughout my candidature. I could not have asked for better supervisors. In particular, I would like to thank Lin for helping me obtain scholarships that have been of great assistance to me over the last few years.

I feel extremely lucky to have met Lin five years ago, where she gave me the opportunity to work as a vacation scholar and introduced me to the area of software agents. Since then my interest in the area has only grown stronger. More importantly, since then I have worked with a great bunch of people.

I would like to thank RMIT for offering me a scholarship to do my PhD and for all the resources made available to me throughout my candidature. I also thank the various technical support staff and administrative staff that have assisted me at different stages of my candidature.

My research was also supported at various stages by Agentis International, Agent Oriented Software (AOS) Pty. Ltd. and the Australian Research Council (ARC) (under grants C49906798 and CO0106934). In particular I would like to thank AOS for providing a free license to use JACK for educational purposes.

A special thank you to my parents who have supported my education all my life. I am very grateful for the enormous amount of sacrifices that they have made in their own lives to allow me to get this far in my education.

I would like to thank Deborah Goldwasser, Smeeta Lalchandani, and Dr. James Harland who have assisted me by providing feedback on my writing at various stages. I would also like to thank Deborah for her moral support and encouragement over the last few years.

Finally I would like to thank my research buddy, David Poutakidis, and my cricket club, whom without I may have gone insane over the last few years.

Thanks to you all!

Dedication

To the loving memory of my grandfather, Hector Mathiasz (9.10.1919 - 13.08.2002).

Contents

List of Figures

Abstract

An important feature of intelligent agents, is their ability to pursue multiple goals in dynamic and complex domains. An "intelligent" agent should be sensible in the way it pursues these multiple goals, in particular when they are pursued concurrently. For example, it is not sensible for a Mars rover agent to attempt a goal to roam the south-side of Mars and another to roam the north-side at the same time. As trivial as this may be, current agent systems do not support this kind of goal related reasoning within their basic infrastructure.

In this thesis, we develop mechanisms that manage the interactions between goals of an agent, in a domain independent way, that can be integrated into the infrastructure of an agent development system. The algorithms we develop are simple and efficient, which makes them highly amenable to practical systems.

There are three main types of reasoning that we address in this thesis. We reason about: resource related conflicts, interference that can occur between goals when goal-steps are interleaved, and situations where one goal can assist in the achievement of another.

Experimental evaluations show that the benefits of our reasoning methods are significant, even in situations when we least expect it, and the cost is negligible, even in situations where the algorithms are stress-tested. This extremely low computational overhead is important, as the agents we develop operate in environments where they deliberate and act in a timely manner. Therefore, integrating these reasoning mechanisms into practical agent systems allows building agents that are more sensible and productive in the way they pursue concurrent goals, without sacrificing their efficiency.

Chapter 1

Introduction

In today's modern world, where technology is constantly evolving, intelligent agents are gaining wide acceptance as a commercially viable and innovative technology. They are used to develop applications in a wide variety of domains (Jennings & Wooldridge 1998, Aylett, Brazier, Jennings, Luck, Preist & Nwana 1998, Georgeff & Rao 1996), ranging from small personal digital assistants to large scale air traffic control systems such as that which was successfully tested at the Sydney International Airport (Ljungberg & Lucas 1992).

The main advantages of intelligent agents are their autonomy and their ability to operate in dynamic, complex domains. Autonomous agents need not be told how to perform a task. Rather, the agent is programmed with multiple possibilities of achieving tasks, and when given a task to be achieved, the agent chooses its own suitable way of achieving the task depending on the situation. When agents operate in highly dynamic domains, where there are a lot of complex interactions, failure is always possible. In such domains, intelligent agents have the ability to recover from failure and, when possible, to avoid it.

Avoiding conflicts however, can be difficult as there are very many different ways in which the agent may choose to accomplish its goals. The choice cannot be pre-determined as the domains are dynamic. This thesis proposes mechanisms for maintaining summarised information about all possible choices that an agent may pursue. This summary information is then used by the agent to reason about potential conflicts, or other goal interactions.

Most agent applications require an agent to satisfy multiple goals (tasks) and often to do so concurrently. In pursuing concurrent goals if the agent does not reason about goal interactions, it is possible for an agent to act in such a way that it causes itself to fail in achieving its goals. For example, consider an autonomous

Mars rover agent[1] that is pursuing its goal to refill its fuel tanks. If it has another goal to transmit some information to earth, it is quite feasible that this could be done concurrently. If however, the data transmission goal required the rover to return back to the lander, then attempting to do this concurrently could well lead to problems. Moving away from the pump prematurely would cause the refuelling goal to fail, and potentially, also cause other goals to fail due to inadequate fuel.

The work we have done enables an agent to reason about these kinds of situations. Using such reasoning it is able to avoid decisions that later (may) lead to failure, or take decisions that lead to greater efficiency. All reasoning is done using summarised information that represents in a concise way, the full space of possibilities. We develop three specific forms of such reasoning, all focusing on the interactions that arise when pursuing multiple goals concurrently. The proposed mechanisms manage the interactions between concurrent goals, avoiding conflicts and exploiting situations where the pursuit of some goals may assist in satisfying other goals. A key feature of the data structures and algorithms developed is their simplicity, which makes them highly amenable to practical systems.

Many modern agent development systems, such as JACK (Busetta, Rönnquist, Hodgson & Lucas 1998), JAM (Huber 1999), dMARS (d'Inverno, Luck, Georgeff, Kinny & Wooldridge 2004) and PRS (Ingrand, Georgeff & Rao 1992), which are based on the popular and well developed Belief Desire Intention (BDI) model of agency (Rao & Georgeff 1991, Rao & Georgeff 1992, Rao, Georgeff & Kinny 1996), are well suited to develop applications where agents operate in dynamic environments. This is mainly due to the mechanisms that handle failure, which are integrated into the framework of these systems. These systems, like many others, allow an agent to pursue multiple goals simultaneously. However, little work has been done on managing the possible interactions between these parallel goals both in the theory of intelligent agents and their practical implementations.

There are a number of ways in which concurrent goals of an agent can interact, some of which are outlined below.

Resource conflicts

An agent in many applications is resource bounded. That is, it has a limited quantity of any given resource. When an agent pursues multiple goals that compete for the same finite resource(s), conflicts are likely to occur. For example, if the agent has 100 units of energy, one goal that requires 70 units of energy and another goal that requires 80 units of energy, then the agent cannot satisfy both goals. If

[1]Examples based on autonomous planetary rovers have been used by many research groups (Clement, Barrett, Rabideau & Durfee 2001, Estlin, Gray, Mann, Rabideau, Castaño, Chien & Mjolsness 1999, Matthies, Gat, Harrison, Wilcox, Volpe & Litwin 1995).

the two goals are pursued in parallel, there is a possibility that both goals will fail (for example, if the first goal consumed 60 units of energy and the other consumed 40 units, of the 100 units available). Consider another example, where the agent has two goals that both require a communication channel, and the agent has only one communication channel that cannot be shared. In this case, whilst both goals cannot be executed in parallel, it is possible to execute one after the other since the communication channel may be re-used by the second goal after the first goal completes. A rational agent should be able to reason about such conflicts with respect to resources.

Interference due to dependencies

An agent pursuing multiple goals may also encounter the following problems. Typically, an agent executes a number of steps in order to achieve some goal, and when pursuing two goals in parallel, the steps of each goal are interleaved. This interleaving could potentially cause problems. For example, consider an agent that has a goal to analyse a soil sample on the south-side of Mars, which requires the agent to go to the south-side, collect a soil sample and perform an analysis. If the agent has another goal to analyse a soil sample from the north-side, it is not sensible for the agent to go to the south-side, and prior to collecting a soil sample, to interleave and head towards the north-side. It is important for an agent to be sensible in the way it interleaves between goals, and recognize when certain conditions must be maintained and protected in particular situations (such as remaining at the south-side until a soil sample is collected).

Positive interactions

While managing conflicting interactions between goals is critical for the correct execution of an agent, identifying and facilitating situations where the steps of one goal assist in achieving another goal, is also useful to allow the agent to function more efficiently and intelligently. For example, if the agent has a goal to analyse a soil sample at the south-side of Mars and another goal to analyse a rock sample also at the south-side, it could potentially start at the lander, go to the south-side, analyse a soil sample, come back to the lander and then go back again to the south-side to analyse a rock sample. A more sensible approach would be for the agent to go the south-side, analyse a soil sample, analyse a rock sample, and return back to the lander. This would require the agent to identify situations where there are common steps between goals that could potentially be merged into a single step (such as going to the south-side).

Existing work

The support for any form of goal related reasoning is limited and varied in current agent development systems. PRS by SRI International (Ingrand et al. 1992) and dMARS by the Australian Artificial Intelligence Institute (d'Inverno et al. 2004) represent goals as events with no internal structure. Other systems like JACK by Agent Oriented Software Pty Ltd. (Busetta et al. 1998) allow various information to be stored within the goal/(event) structure, but the goals are transient in that if they cannot be pursued as they occur they are ignored. On the other hand, Huber's JAM (Huber 1999) keeps track of goals that cannot be immediately pursued for a later attempt. JAM also uses the goal structure to reason about when goals are satisfied. Irrespective of how they represent goals, none of these systems support the management of interactions between concurrent goals.

In the theory of intelligent agents, there has been some work on the formal properties of goals (Hindriks, de Boer, van der Hoek & Meyer 2000, Bell & Huang 1997, Doyle, Shoham & Wellman 1991, van Riemsdijk, van der Hoek & Meyer 2003, van Linder, van der Hoek & Meyer 1995) and an agent's commitment towards achieving its goals (Rao & Georgeff 1992, Cohen & Levesque 1990*a*). However, most of this work is presented within a logical framework that cannot be easily mapped into a practical agent system suitable for large scale, real-time applications. In this thesis, we provide programming level semantics of the reasoning mechanisms that we describe, which allow them to be easily integrated into a practical system.

There has been a significant amount of work on conflict management (Tessier, Chaudron & Müller 2000) and positive interactions (Cohen, Levesque & Smith 1997) between agents. These methods rely on knowledge that is shared and/or communicated amongst the agents. While some of these methods could be applied to managing conflicts between goals of a single agent, far better solutions can be developed for this purpose. The primary reason for this is because the algorithms that perform such reasoning will have direct access to the agent's mental state, providing more informative, reliable and up-to-date information.

Summary of contributions

This thesis provides specific data structures and algorithms that support reasoning about three particular kinds of interactions between goals[2] a single agent:

1. resource related conflicts;

[2]We only consider goals where the agent has to achieve some state of the world, and do not consider other types such as goals that require a state of the world to be maintained. Further discussion on this issue is provided in section 2.1.

2. interference that can occur when goal-steps are interleaved; and
3. situations where one goal can assist in the achievement of another.

Each of these mechanisms perform reasoning based on summarised information regarding all possible options that an agent may pursue. This summarised information is created at compile-time and updated continually at run-time as choices are made, and possibilities are narrowed down. The reasoning methods are domain independent, and can readily be integrated into BDI agent systems as part of the infrastructure support. This is shown by our experimental work that evaluated our algorithms, where we extended JACK (Busetta et al. 1998), a BDI agent development toolkit, to incorporate the reasoning methods developed in this thesis. Empirical results indicated that the computational overhead of using our reasoning in practical systems, even in situations where there are an extremely high number of interactions between goals, is negligible. In contrast, the benefits are evident even in situations where there are low levels of interactions between goals. This makes the algorithms well suited for complex, time critical, and dynamic environments.

Outline of thesis

The rest of this thesis is organized as follows. In chapter two, we provide background information which details the concepts of an agent, goals, plans, the BDI model, and the execution model of the agents described in this thesis. This chapter also includes work related to this thesis, identifying similarities and differences between existing work and the work presented in this thesis.

In chapter three, we present our structures for goals and plans. We discuss how our representation satisfies the properties of goals and compare it to the representations in current practical implementations. We end chapter three by discussing the kinds of reasoning that we perform about goal interactions.

In chapter four, we address the management of resource conflicts, providing all the necessary data structures and algorithms required to identify such conflicts, and discuss ways in which they could be resolved. Algorithms that identify interference when goal steps are interleaved are presented in chapter five, along with mechanisms that monitor and schedule goal and plan execution, such that interference (if any) is avoided. Having dealt with negative interference, in chapter six we reason about positive interactions. We present algorithms that identify common plans between goals that could possibly be merged, and also present algorithms for monitoring and scheduling goal and plan execution to facilitate plan merging.

Chapter seven presents the details of the experimental evaluation and the results obtained. Finally, conclusions and possible future research areas are presented in chapter eight.

Chapter 2

Background

This chapter reviews past literature and introduces the concepts of agents, goals, plans and the Belief Desire Intention (BDI) model, all of which are necessary in understanding this thesis. Section 2.4 of this chapter also explains the choice of Jack Intelligent Agents$^{\text{TM}}$ to perform the experimental work for this thesis. Whilst few studies have focused upon managing interactions between multiple goals of a single agent, a number of studies have researched the management of interactions between agents, as well as in the planning domain. Of particular relevance is Clement et al.'s research on co-ordinating Hierarchical Task Network (HTN) planners. HTN planners are introduced and compared to BDI systems in section 2.5, which is necessary to appreciate the similarities and differences between the work of Clement et al. and some of the work presented in this thesis. Literature related to conflicts and positive interactions are reviewed in sections 2.6 and 2.7 respectively.

2.1 Agents and Goals

The dictionary definition of an agent is an entity that acts on behalf of another. A software agent is a computer program that performs tasks on behalf of humans or other software agents, with certain characteristics that distinguish them from other software. There are many descriptions of these characteristics required by a software program to be qualified as an "agent" (Franklin & Graesser 1997, Georgeff, Pell, Pollack, Tambe & Wooldridge 1999, Maes 1994, Wooldridge & Jennings 1995). Amongst these, the description of Wooldridge and Jennings (1995) is commonly used. They require an agent to have the following properties:

- *autonomous*: agents operate without the direct intervention of humans or other agents, and have control over their actions and internal state;

- *social*: agents interact with other agents (and possibly humans) via some kind of agent-communication language (Wooldridge 2002, chapter 8);
- *reactive*: agents respond in a timely fashion to changes that occur in their environment;
- *pro-active*: agents show goal-directed behaviour; and
- *situated*: agents are placed in, and interface with an environment.

It is important for an agent to be both reactive as well as pro-active. For example, a pro-active soccer playing agent may move to a position with the goal of marking an opponent (providing a defence). If the ball appears in its path it should behave reactively, taking control of the ball. This in turn, may cause the agent to be pro-active in forming a goal to score against the opponent.

The types of agents that we focus upon are those autonomous agents that explicitly represent mental attitudes, in particular beliefs, plans and goals; and those that perform reasoning about their goals. While these agents are social and reactive, these aspects are not explored in this thesis. The focus is on the autonomous, pro-active behaviour of a *single* agent that is situated in a particular environment.

A central concept for an agent to exhibit pro-active behaviour is the concept of goals. In the context of this thesis, a *goal* is a state of the world that the agent wants to achieve. Once the state has been achieved the goal will no longer be desired. For example, a goal to go to the beach requires the agent to achieve the state of being at the beach. When the agent arrives at the beach the goal is satisfied. These type of goals are termed *achievement goals*.

There are other types of goals (Braubach, Pokahr, Moldt & Lamersdorf 2004) such as *maintenance goals*, which require the agent to maintain a particular state of the world for a given period of time. For example, a goal to remain at the beach for an hour is a maintenance goal.

This thesis reasons about achievement goals as they are the most common type of goals in practical applications. It is first necessary to understand the mechanisms for reasoning about this goal type, before the reasoning can be extended to other goal types. While some of the algorithms presented could be easily modified to accommodate other goal types such as maintenance goals, other algorithms presented are more complex and further work is required to extend them to accommodate other goal types.

Some properties of goals that are commonly identified are as follows:

Persistent (Rao & Georgeff 1992, van Linder et al. 1995, Bell & Huang 1997, Hindriks et al. 2000) :

This property requires an agent to persist in its goal achievement until the goal is satisfied, or there is a justifiable reason to end its pursuit.

In their logic, Rao and Georgeff (1992) do not specify when an agent should relinquish its goals. However, they define various commitment strategies which determine the level of persistence of goals (see section 2.3 for details).

Hindriks et al. (2000) adopt the *blind commitment* strategy as defined by Rao and Georgeff (1992), where the agent drops a goal if and only if it believes the goal has been achieved.

van Linder et al. (1995) form commitments towards a goal (or a wish as they call them), and drop these commitments if the agent believes them to be fulfilled, or no longer feasible.

Bell and Huang (1997) require the agent to persist in achieving its goals, until the agent's beliefs are revised such that the goal is no longer necessary.

Unachieved (van Linder et al. 1995, Hindriks et al. 2000) :

An agent should *only* pursue a goal if it is not already achieved. In the event that the goal is achieved, it should be dropped with success. This allows the agent to avoid wasting time and/or resources. Note that this property is only true for achievement goals and may not hold true for other goal types. In particular, it is not true for maintenance goals as they require conditions not just to be achieved, but also to be maintained for a period of time.

Possible (Rao & Georgeff 1992, van Linder et al. 1995, Bell & Huang 1997) :

Rao and Georgeff (1992) require goals to be possible in that the agent must believe that it can achieve its goals. While this property does not guarantee that the goal will succeed, it ensures that the agent does not take on goals that it believes it cannot satisfy.

Bell and Huang (1997) call this property *realisable*, which requires the agent to have some plan to achieve a goal at the time that it adopts the goal.

van Linder et al. (1995) require goals to be *implementable*. That is, it must be possible for the agent to perform a sequence of actions which result in the goal being satisfied.

Consistent (Rao & Georgeff 1992, Bell & Huang 1997) :

Goals are required to be *consistent* in that they must be jointly realisable with other goals of the agent. Therefore, an agent should not simultaneously pursue a goal that it believes would conflict with another goal. For example, if an agent has a goal of exploring Mars and another goal of exploring the

Moon at the same time, then the agent should recognize the conflict and resolve it in favour of the more important goal.

Bell and Huang (1997) term this property *coherence*, and enforce this in their logical framework.

Rao and Georgeff (1992) constrain goals to be consistent but do not provide a detailed definition of consistency, nor mechanisms that recognize inconsistent goals.

Known (Rao & Georgeff 1992) :

A rational agent should know what goals it has; in other words, goals should be *known* to the agent. This is an essential pre-requisite for the agent to be able to reason about interactions between goals as well as to be accountable for its actions in relation to the goals.

Permissible (Bell & Huang 1997) :

Bell and Huang (1997) also require goals to be *permissible* in that the agent should not violate any legal or moral constraints that may currently apply.

Goals are an integral part of a large body of theoretical work. However, there is only a limited amount of work that provides an explicit structure for a goal that can be directly implemented in a BDI-like agent system. Some of the relevant existing work is described below, first with respect to the theory of agents and then with respect to the practical implementations of these theories.

Hindriks et al. developed an agent programming language called GOAL (Goal Oriented Agent Language) (Hindriks et al. 2000). In contrast to other attempts (Shoham 1993, Wobcke 2000) to bridge the gap between logical formalisms and a programming language that realises the logic, they provide a logical formalism, a programming language, and a set of formal semantics that relates the logical formalism to the programming language. In their GOAL language they incorporate the concept of a declarative goal in the form of the end state of the goal that the agent wants to reach. This *end state* is represented as a statement of a logical language. For the purpose of their work, they assume this logical language to be a propositional language. Their logic requires that goals not be entailed by beliefs (i.e. they are not already achieved) and that goals be satisfiable. In contrast to the work by others, they do *not* require that a set of goals $\mathcal{G}$ be consistent. For example, $\mathcal{G}$ can contain both p and $\neg p$. They handle this conflict by allowing goals to be achieved in sequence rather than simultaneously. For example, if $\mathcal{G} = \{p, \neg p\}$ then p can be achieved and relinquished, and $\neg p$ can be achieved at a later time. In this thesis we provide an explicit representation for goals that not only captures the end state of the goal but also captures further properties that are required to reason about goal interactions. The agents that we develop attempt to achieve goals simultaneously when possible. Even if the end states conflict, if there are intermediate steps that can be performed in parallel then the agent will attempt to do so. This is unlike the GOAL framework where the goals are executed completely in sequence. The GOAL framework also does not support the nested parallel and sequential structures of sub-goals and atomic actions within a plan that is possible in modern BDI systems, and that is supported within the framework developed in this thesis.

Bell and Huang define and formalise the notion of goal hierarchies based on preference (Bell & Huang 1997). In their work an agent has a set of goals, and considers some of these goals to be more important than others. This preference amongst goals defines a goal hierarchy at any given point in time. This hierarchy is not static since the preferences between goals are dynamic by nature. For example, a goal to go to the beach may take preference over a goal to watch a movie on a warm sunny day, but if it rains the agent may change its preference towards the goal of watching a movie. They define formally how goal preference can be used to revise goals in a rational manner as circumstances change. Although dynamic, the goal hierarchy is stable in that an agent's preference towards its goals does not change unless a relevant incident occurs. In the previous example, the relevant incident that caused the change in preference was the change in the weather. This stability increases up the goal hierarchy, which means that goals with higher preference are more stable. An important property of the goals within the goal hierarchy is that they are coherent and jointly realisable. The agent will not adopt a goal that conflicts with another goal of higher preference. Their work provides the semantics of preference and coherence between goals but does not provide detailed algorithms on how preference can be derived, or how conflicts can be recognized between an agent's goals. While preference of goals can be handled in a simplistic manner by providing a utility value to each goal, conflict management is not trivial. This thesis provides detailed algorithms that manage conflicts. Although goal preference and goal revision based on preference is sensible, data structures and algorithms that facilitate such reasoning are beyond the scope of this thesis.

Doyle et al. (1991) in their work on 'a logic of relative desire', present a formal logic that provides a framework for representing and reasoning about desires. The main focus of their work is on the ability to capture in the logic the relative nature of a desire. For example, the desire to wear a raincoat is likely to be relative to whether or not it is raining. Similar to the work of Hindriks et al. (2000), they too represent goals as propositions. They provide a logic that considers relative desire between arbitrary conditions expressed in a propositional language, provide a set of inference rules that support reasoning about relative desire, and show how relative desire varies with restrictions imposed by beliefs. The representation of goals and plans and the reasoning algorithms presented in this thesis capture an implicit notion of relativity between goals, although not formally defined nor fully developed. For example, an agent would decide not to pursue a goal that it would normally pursue, if that may cause conflict with other goals. Goals are also constrained by beliefs in that plans that achieve a goal have a context condition attached to them that determines their applicability in a given context.

Most of the approaches in representing goals only represent the declarative aspect of the goal (i.e. the end state to be achieved) and not the procedural aspect (i.e. what plans can be used to achieve the goal), until the more recent work done by our group (Winikoff, Padgham, Harland & Thangarajah 2002). In that work, we argued that both the procedural and declarative aspects are important. We provided an explicit structure for goals based on their desired properties and provided operational semantics that realised these properties. A goal is defined as a 3-tuple $G =< s, P, f >$ where s defines the conditions to be achieved by the goal, f is the failure condition which defines when the agent should no longer pursue the goal and P is the set of possible plans that could be used to achieve the goal. While a discussion on the type of reasoning that can be performed about goals is presented, the details and algorithms of this reasoning are not provided. The representation of goals we present in this thesis also captures the procedural and declarative properties of goals, as well as further properties that are necessary to reason about goal interactions.

van Riemsdijk et al. (2003) also attempt to incorporate both the declarative and procedural aspect of goals into their *Dribble* programming language. This work is more recent but similar to that in Winikoff et al. (2002). They represent goals and beliefs as propositional formulae, and provide operational semantics for mental state changes of the agent and the execution of goals. In their work, they use goals for plan selection only and do not perform any reasoning about goal interactions.

Inspired by Dribble (van Riemsdijk et al. 2003), Dastani et al. (Dastani, van Riemsdijk, Dignum & Meyer 2003, Dastani, de Boer, Dignum & Meyer 2003) extended the agent programming language 3APL (An Abstract Agent Programming Language) (Hindriks, Boer, Hoek & Meyer 1999) to include the declarative aspect of goals. In this extension they improve on Dribble in two ways:
(i) In Dribble the declarative goal was used only for plan selection, while in the new 3APL language, goal-rules can be specified to reason about goals; and
(ii) Dribble is a propositional language without variables, which limits its programming power. In 3APL however, the beliefs and goals of an agent can be represented by a first-order domain language, which is more expressive.
The initial 3APL programming language (Hindriks et al. 1999) represented goals as procedures for the agent to execute. In the extended 3APL (Dastani, van Riems-

dijk, Dignum & Meyer 2003), goals represent situations that the agent wants to realise, and plans are introduced as procedures to allow the agent to satisfy these goals. 3APL is a rule-based programming language and in the extended 3APL, the rules can be categorized as *goal rules*, *interaction rules* and *plan rules*. *Goal rules* are used to generate, revise, or drop goals. *Interaction rules* are used to generate plans that achieve goals (they specify the conditions that the agent must believe to be true in order for it to choose a particular plan to satisfy a goal). *Plan rules* are used to revise and drop plans. In the initial 3APL system, only plan rules were included and they were called *practical reasoning rules*. In order to determine the flow of control among the various 3APL components, an *interpreter* is required. This interpreter determines the order in which rules should be applied, when actions should be executed, and when beliefs should be updated. 3APL allows the design and implementation of an agent system to be specified in terms of mental attitudes such as beliefs, goals and plans, where it is possible to verify whether the implementation matches the specification. Although Dastani et al. (2003) include declarative goals and goal-rules that allow some reasoning about goals, this reasoning is limited. They do not provide any mechanisms for managing interactions between goals.

The formalisation of van Linder et al. (1995) defines *wishes*, *goals* and *commitments*. *Wishes* (similar to desires) are what the agent would like to achieve. *Goals* are wishes that are unfulfilled and which the agent has some means of achieving. An agent may form *commitments* towards the plan of achieving a goal and may also drop these commitments if the agent knows the commitments are no longer feasible. In this thesis, we similarly require goals to be unachieved and possible. Goals that cannot be achieved immediately, but could possibly be pursued at a later stage, are stored and re-tried when possible, much like a wish. It is important to note that in their approach and ours, the agent does not store goals (wishes) that it knows can never be achieved. Commitments are implicit in the framework that we present in this thesis, and an agent's commitment towards its goals is such that once a goal is adopted, it is only given up if either the goal is satisfied, or if it conflicts with another preferred goal, where the only means of conflict resolution is to end the pursuit of the goal.

In practical implementations of agent development systems where the agents are built on mental attitudes such as beliefs, goals, plans (intentions) and actions, an explicit representation of goals is absent except in JAM (Huber 1999) and 3APL (Dastani, van Riemsdijk, Dignum & Meyer 2003). As we have discussed 3APL before, we do not include it in the discussion below.

Systems like Agent-0 (Shoham 1991), AgentSpeak(L) (Rao 1996), PRS (Ingrand et al. 1992), dMARS (d'Inverno, Kinny, Luck & Wooldridge 1998) and JACK (Busetta et al. 1998), do not represent goals explicitly, but instead capture goals *implicitly*.

Agent-0 represents goals as a set of *commitment rules*, where each contains a message condition, mental condition and an action. When a message that matches the message condition arrives and the mental condition is satisfied, the associated action will be executed. AgentSpeak(L), PRS, dMARS and JACK all represent goals as an *event* that triggers the agent to select a particular plan to execute. If there is no current plan applicable to handle the event, the event is discarded.

JAM (Huber 1999) represents goals explicitly in the form of:

$\langle$ *goal_type*, goal_name, *parameter1* . . . *parameterN* : *utility* $\rangle$

The *goal_type* can be one of *maintain*, *achieve* or *perform*. The *goal_name* is a label that identifies the goal and the *parameter*s are the arguments for the goal. The *utility* is an optional numerical field that indicates the importance of the goal which is used to ensure that the agent always pursues higher preference goals first. The following are example goals of an agent:

perform wander_lobby;
maintain at_lobby;
achieve at_lobby 50;

An *achieve* goal requires the agent to perform an action to achieve a condition such as to be at the lobby. A *maintain* goal requires the specified condition to be re-attained if it ever becomes false. A *perform* goal specifies an action that the agent must execute, and is not concerned with achieving or maintaining any conditions. Similar to the approach in this thesis, JAM ensures that top level goals are persistent until they are satisfied either by successful plan execution or opportunistically by some other means, such as by another agent. It is the explicit structure for goals maintained by the agent that allows it to identify when a goal is successful with respect to achievement goals and when it is violated for maintenance goals.

The goal structure also stores goals for later attempts, thus overcoming the limitation of systems like JACK, where an event is discarded if no applicable plan currently exists.

Unlike top-level goals, sub-goals are not persistent in the JAM model. In the event that they fail, the agent does not attempt to find other means of satisfying them. In our work however, we treat sub-goals as similar to top-level goals, and require a similar level of commitment from the agent towards them. The representation for goals in the JAM model is sufficient for the reasoning that it performs about the agent's goals. However, in order to perform more complex reasoning such as managing the interactions between goals as we have done in this work, a more expressive representation is required and is presented in the chapters ahead.

Where do goals come from?

Many researchers have attempted to answer the question, “where do goals come from?”. Dignum and Conte look at goal formation in terms of built-in goals and *instrumental reasoning* (Dignum & Conte 1997). Built-in goals are formed when the agent is developed, and are often not explicit. For example, a Mars rover agent may have a built-in goal of *maintaining a high energy level*. *Instrumental reasoning* occurs when an agent forms goals using built-in goals and some *instrumental beliefs*. An *instrumental belief* indicates to the agent how a certain condition contributes to achieving an original goal, which can be seen as the reason for a new goal. For example, if the built-in goal is to maintain a high level of energy, an instrumental belief could be that refuelling periodically is instrumental to maintaining high levels of energy. This may cause the agent to form a new goal of refuelling every hour.

Mohanty et al. describe how influencing can be used as a strategy for goal adoption in a multi-agent setting (Mohanty, Patra & Naik 1997). The key idea is for one agent to communicate in such a way that it causes change to either the beliefs, desires or intentions of another agent, which in turn could lead to new goals being formed by the other agent.

Norman and Long use the concept of a *motive* that triggers goal creation (Norman & Long 1995). A motive is defined as a need that causes the agent to act. For example, a motive of hunger would cause the agent to form a goal of eating some food.

Later work by Luck and d’Inverno (1998) describes an agent framework that also uses *motivation* for goal creation and adoption. In more recent work, they, together with Munroe, build on their former work to evaluate the *worth* (utility value) of a goal based on motivations (Munroe, Luck & d’Inverno 2003*b*), and also use motivations to evaluate the worth of a given environmental state to a goal state (Munroe, Luck & d’Inverno 2003*a*).

In this thesis we are not concerned with the theory of goal formation. We assume that goals are programmed into the agent and pursued by the agent when it is made active, or when it is triggered by a human user via a program interface or by another agent via some communication medium.

2.2 Agent Architectures

There are many architectures that have been proposed for building intelligent agents (Wooldridge & Jennings 1994). The architecture that serves as a basis for the agents in this thesis is the Belief Desire Intention (BDI) model of agents and is discussed in the next section. Apart from the BDI model, Brooks’ *subsumption ar-*

chitecture (Brooks 1986), Chapman and Agre's model of agents illustrated by their *PENGI* system (Agre & Chapman 1987, Chapman & Agre 1987), Rosenschein and Kaelbling's *situated automata* (Rosenschein 1985, Rosenschein & Kaelbling 1986), the *agent network architecture* by Maes (Maes 1989, Maes 1991), and InteRRaP by Müller et al. (Müller 1994, Müller & Pischel 1994) are also well known.

Rodney Brooks, whose main area of research focussed on autonomous mobile robots, argued that intelligence can be achieved without explicit representation of mental attitudes or the environment. He proposed an architecture for building agents (Brooks 1986, Brooks 1991*b*, Brooks 1991*c*, Brooks 1991*a*), where an agent was defined in terms of a hierarchy of task achieving *behaviours*. A *behaviour* specifies a complete cycle from perception to action for a very specific task. For example, an *avoid obstacle* behaviour is one that prevents the agent from running into, or being hit by an obstacle. Behaviours at higher levels of the hierarchy subsume those at lower levels. For example, an *explore room* behaviour may subsume the *avoid obstacle* behaviour. Some of the main features in this approach are that:

- there is no need for careful planning (at run-time) as the behaviours are prespecified;
- since the agent interacts directly with the environment, it is not necessary to make assumptions about the environment; and
- the agent uses the real world as its own, which avoids the need for internal model building.

The resulting systems based on this architecture are extremely simple in terms of the amount of computation needed. It has been successful in some applications, which is evident from the number of robots that have been built by Brooks using this architecture, including six-legged walking robots, vacuuming robots and robots that collect cans. However, the lack of internal representation makes it difficult to reason about interactions between behaviours, and engineering large scale applications can be very complicated.

Pattie Maes developed an agent architecture where an agent is defined as a set of *competence modules* (Maes 1989, Maes 1991), which resemble the operators found in classical STRIPS style planning (Fikes & Nilsson 1990). Each module has pre-conditions that define the conditions that must be achieved before the module can begin execution, and effects which are specified in terms of an *add-list* and a *delete-list* of beliefs. These modules also contain an *activation level*, which provides an indication of the relevance of the module in a particular situation. The activation level of a particular module varies depending on the situation that the agent is in. The concept of a network arises in the way modules are connected to each other. One such form of connection is when the pre-conditions of one module can be achieved by the post-conditions of another module. The way modules are connected influences their respective activation level. A difference and an advantage between this approach proposed by Maes compared to Brooks' subsumption architecture (Brooks 1986), is that the control flow is not pre-specified but is determined at run-time, and there is a declarative aspect to goals within the system. However, this approach is linear and does not allow for hierarchical structures. For a detailed analysis of this architecture refer to the work by Tyrrell (1994).

Agre and Chapman developed an architecture for developing agents (Chapman & Agre 1987) which, like Brooks' subsumption architecture, is reactive. They argue that much of everyday behaviour is spent in *generic routines*, specifically *see, react, repeat*. For example, notice an obstacle, step around it; hear a sound, turn to look; taste something bad, spit it out. The behaviour is therefore directly coupled to the input from the environment and is purely reactive. Similar to Brook's approach (Brooks 1986), there is no need for internal modelling of the world nor reasoning about actions. Agre and Chapman illustrate this approach by developing an arcade game *PENGI* (Agre & Chapman 1987), where the environment is simulated and dynamic. This approach works well for systems like *PENGI*, where the actions of the agent can be purely driven by percepts. However, the problem with this approach is that it does not allow for goal-directed behaviour, where the actions are driven not merely by percepts, but by high level goals. Take for example, a goal to pick up a block that is hidden behind a wall. A reactive agent that cannot see the block will not respond, but a deliberative agent would reason about this problem using its internal representations and act accordingly.

Situated Automata proposed by Rosenschein and Kaelbling (1985, 1986) is another example of a reactive architecture. It is defined for agents that are situated in one particular environment, with an emphasis on the interactions between the agent and the environment. There are two components in their approach: a *perception* component and an *action* component, which are specified in declarative terms. Once specified, these components are compiled into an automaton, which satisfies the declarative specification. A key feature of this approach is that the combined automaton (i.e. perception and action) can operate in a provably time-bounded fashion, mainly because it does not do any symbolic reasoning.

InteRRaP by Müller and Pischel (Müller 1994, Müller & Pischel 1994) is a *hybrid architecture* that is neither purely reactive nor purely deliberative. It consists of three layered components: a behaviour layer, a plan layer and a co-operation layer. These layers are embedded in a control framework that mediates between the layers. The *behaviour layer* implements and controls the reactive capability of the agent, interacting with the *world interface* that deals with perception, action and communication. The *plan layer* generates plans for a single agent in response to requests from the behaviour layer. The *co-operation layer* generates joint plans that satisfy goals of a number of agents, in response to requests from the plan layer. These last two layers form the deliberative component of the agent. The InteRRaP architecture is similar to TouringMachines by Ferguson (1992). An important factor in such layered hybrid architectures is controlling the interaction between the layers and maintaining overall coherence of behaviours between the layers.

2.3 The Belief Desire Intention Model

The Belief Desire Intention (BDI) model is well developed and widely used in modern agent development systems. Agents that are built based on the BDI model are referred to as *BDI agents*. The mechanisms developed in this thesis are suitable for, but not restricted to, the agents implemented in BDI systems.

The philosophical work of Bratman (1987, 1988, 1999), formalised by Rao and Georgeff (1991, 1992) provides a basis for much of the work in BDI agents. BDI agents are characterised by having the mental attitudes of beliefs, desires and intentions, described in detail below:

Beliefs can be described as the information that an agent has about itself and its environment. For example, the agent may believe that it is at the beach (belief about itself) and also believe that it is night time (belief about its environment). The term "belief" is used, as opposed to "knowledge", because the agent's beliefs are not necessarily true facts, but are believed to be true by the agent. The *T* axiom of modal logic (Kripke 1986) is often used to distinguish knowledge from belief and it states that *what is known is true*.

Beliefs can change over time and *belief revision* (Gärdenfors 1992, Nebel 1989) (or *belief change* (Booth, Chopra, Ghose & Meyer 2003, Ghose, Hadjinian, Sattar, You & Goebel 2004, Booth, Chopra, Meyer & Ghose 2004, Meyer, Ghose & Chopra 2001)) deals with changes in the beliefs of an agent. One of the most influential approaches to belief revision is the AGM theory developed by Alchourrón, Gärdenfors and Makinson (1985, 1988, 1992). Their framework focuses on the process through which beliefs are added and/or retracted from belief states whilst maintaining consistency.

Later work (Katsuno & Mendelzon 1991, Winslett 1990) identified two parts to belief revision: *update* and *revision*. They differ in that belief *revision* occurs when the agent realises that its beliefs are incorrect or incomplete compared to the real world model, and *update* is triggered by actual changes in the world which render the agent's beliefs obsolete, forcing it to change them.

While the abstract BDI interpreter developed by Rao and Georgeff (Rao & Georgeff 1995) includes an implicit notion of belief revision, no description of the process of belief revision is described. In implemented BDI systems like JACK, dMARS and PRS, beliefs are *updated* rather than revised. Beliefs are updated in such systems when there is change due to actions performed by the agent (e.g. the agent uses energy), or due to changes in the environment perceived by the agent (e.g. night turns into day). In these systems the necessary belief updates must be explicitly programmed into each application as there are no in-built mechanisms that support reasoning about beliefs, such as recognizing inconsistent beliefs and reasoning about the logical consequences of beliefs.

Desires are what an agent would like to satisfy. Although desires are conceptually related to goals within the BDI framework developed by Rao and Georgeff, desires are allowed to be inconsistent, whereas goals are required to be consistent and believed to be possible by the agent (Rao & Georgeff 1991, page 474). This distinction is also made by Cohen and Levesque (1990*b*). For example, an agent could have a desire to go to the beach on Saturday as well as a desire to stay home all weekend, but cannot adopt both desires as goals as they are not jointly realisable. An agent could also have a desire to turn night into day, but cannot sensibly have a goal to do so, as it is not achievable[1](Padgham & Lambrix 2000). The goals of an agent are therefore a consistent subset of its desires which it believes to be achievable.

Intentions are formed when an agent commits to achieving a particular goal. Having a goal to go to the beach by itself will not achieve it, if the agent does not intend some means of achieving this goal. This *means* of achieving a goal is a *plan* of the agent.

While there are many views on the concept of intentions as described by Bratman (1999), we do not describe them here, as in this thesis we regard an intention to be an instantiated plan.

A *plan* specifies a procedure for achieving a particular goal. The agent achieves a goal, by adopting a plan that it believes would achieve the goal.

Bratman argues that plans are typically partial and hierarchical (Bratman 1987), which is also reflected in the framework presented in this thesis. For example, the agent may adopt a plan to go to the beach on Saturday without

[1]This assumes that the agent cannot control the environment.

AI1 $GOAL(\alpha) \supset BEL(\alpha)$, where α is an O-formula, that is a well-formed formula that contains no positive occurrence of *inevitable*[2]
AI2 $INTEND(\alpha) \supset GOAL(\alpha)$
AI3 $INTEND(\textit{does}(a)) \supset \textit{does}(a)$, where a is a primitive event[2]
AI4 $INTEND(\phi) \supset BEL(INTEND(\phi))$, where ϕ is a state formula[2]
AI5 $GOAL(\phi) \supset BEL(GOAL(\phi))$
AI6 $INTEND(\phi) \supset GOAL(INTEND(\phi))$
AI7 $\textit{done}(a) \supset BEL(\textit{done}(a))$
AI8 $INTEND(\phi) \supset \textit{inevitable eventually}(\neg INTEND(\phi))$

Figure 2.1: *Axioms for beliefs, goals and intentions by Rao and Georgeff.*

deciding on the exact details of how to get to the beach (i.e. partial), which are left for deliberation at a later more informed stage. This is sensible, because the plan that gets the agent to the beach may depend on a number of factors, such as the amount of money the agent has (to catch a taxi maybe), the weather (to walk) and so on, which cannot be determined until Saturday arrives.

Rao and Georgeff, in their formalisation of the BDI theory (Rao & Georgeff 1991, Rao & Georgeff 1992), define characteristics of goals, providing a set of axioms that they refer to as the *basic I-system* (Rao & Georgeff 1991). These axioms are represented in a logical framework based on modal logic (Kripke 1986). While understanding the logic is not critical to the context of this thesis, the properties of goals, beliefs and intentions defined by the logic are important. These properties are as follows (the logical axioms are shown in figure 2.1):

- If the agent has a goal α, then it must believe α to be an option. Note that this does not necessarily mean that the goal will be achieved, but that it is believed to be possible. (axiom AI1)
- For an agent to have an intention α, α must be a goal of the agent. This states that the agent only forms an intention, if the intention aims to achieve a particular goal. (axiom AI2)
- If an agent has an intention to perform an atomic action *a*, then it will perform that action. This connects intention to actions. (axiom AI3)
- The agent should be aware of its intentions and goals. (axioms AI4 and AI5)

[2]For more details on the logic see (Rao & Georgeff 1991).

- If the agent has an intention to achieve α, then the agent must have a goal to intend α. (axiom AI6)
- When the agent has performed an action, it should believe that it has done the action. (axiom AI7)
- It is inevitable that an agent will eventually drop its intentions (either because they succeed or for some other permitted reason). In other words, the agent will not maintain its intentions forever. (axiom AI8)

While the logical axioms require the agent to eventually drop its intentions (axiom AI8), the circumstances under which the agent no longer intends a particular goal are not specified within this framework. Instead, they allow for these circumstances to be determined by a commitment strategy. Rao and Georgeff describe some commitment strategies in their work: *blind*, *single minded* and *open minded*.

- a *blindly* committed agent maintains intentions until the agent believes them to be true. This commitment strategy is clearly very strong as it means the agent will keep the intentions until it believes them to be true, even if it is impossible to achieve the intentions.
- a *single minded* commitment is when the agent maintains its intentions until they are believed to be true or are believed to be impossible to achieve. While this is less strict that the blind commitment, it still does not allow an agent to re-consider its intentions except on the basis of believing them to be impossible.
- an *open minded* commitment requires the agent to maintain its intentions until they are believed to be true or no longer correspond to goals of the agent. This commitment strategy allows an agent to re-consider its goals but does not require an agent to drop impossible intentions.

From the above commitment strategies, it seems that a sensible commitment strategy would be to adopt a combination of both the *single minded* and *open minded* commitment, where an agent should maintain its intentions until they are believed to be true, are no longer goals of the agent, or are believed to be impossible to achieve.

Further work on formalising intentions and commitment towards intentions has been done by Cohen and Levesque (1990*a*) and also by Padgham and Lambrix (2000). Padgham and Lambrix (2000) introduced the notion of a *self-aware* agent that drops its intentions if it believes that it no longer has the capability for that intention. Cohen and Levesque (1990*a*) argue that intentions need not be explicitly represented and can be described using the base constructs of beliefs and goals. In their work, they define the notion of a *persistent goal* which the agent pursues if it believes it to be possible and not currently true, and drops the goal if the goal is satisfied or believe that it will never be satisfied.

Rao and Georgeff have also described an abstract BDI Interpreter for situated systems that models the BDI architecture, which has been the basis for the execution model of some practical agent systems, such as PRS (Ingrand et al. 1992), dMARS (d'Inverno et al. 1998) and JACK (Busetta et al. 1998). The interpreter is as follows (Rao & Georgeff 1992):

```
" Initialize-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 "
```

The inputs to the system are *events*[3], which are received through the *event-queue*. These events can be either external or internal. External events occur in the environment and may generate internal events that occur within the system. In the above interpreter, events are assumed to be atomic and recognized by the system after they have occurred.

The data structures *B*,*G* and *I*, represent beliefs, goals and intentions respectively. These structures are dynamic and global, and can be queried and updated as necessary. The update operations however, must ensure that the constraints on these mental attitudes, as formalised in Rao and Georgeff's BDI logic, are followed.

In the main interpreter loop given above, at the start of each cycle, the agent examines the event-queue and generates a set of *options* that the agent can pursue in response to each event (step 1). From these options, the agent deliberates on which options to adopt (step 2) and adds these to the intention structure (step 3).

These *options*, in the general case, can be tasks, plans, production rules, finite automata or circuit networks. However, in most practical implementations such as

[3]An *event* is a trigger that requires the agent to respond to it in some way, which could be performing some action or updating its beliefs. For example, the event of running out of fuel may cause an agent to re-fuel.

PRS (Ingrand et al. 1992), JACK (Busetta et al. 1998) and dMARS (d'Inverno et al. 1998), the options are represented as *plans*. A *plan*, as defined by Rao and Georgeff (1995) is an abstract specification of the means of achieving certain desires. A plan consists of a *body* that describes the primitive actions or sub-goals that must be achieved for the plan to succeed. The conditions under which a plan can be chosen as an option (in step 1) are specified by an *invocation* condition and a *pre-condition* for each plan. The *invocation* condition specifies the event that triggers the invocation of the plan, and the *pre-condition* specifies the situation that must be true for the plan to begin execution. When a plan is selected to be executed, the actions/sub-goals specified by the plan body are added to the intention structure (step 3).

In the next step of the cycle, if there is an intention to perform an atomic action at this point in time, the agent performs this action (step 4).

The cycle ends by incorporating any new external events that occurred during the cycle into the event-queue[4] (step 5), and dropping all successful goals and satisfied intentions (step 6), as well as impossible goals and unrealisable intentions (step 7). This maintains goals to be unachieved and possible.

There are many agent development systems that are based on the BDI model, such as PRS (Ingrand et al. 1992), AgentSpeak(L) (Rao 1996), dMARS (d'Inverno et al. 1998), JACK (Busetta et al. 1998), JAM (Huber 1999), Jason (Bordini & Hübner 2004) and JADEX (Pokahr, Braubach & Lamersdorf 2003). From these systems, we chose JACK to test and evaluate the data structures and algorithms that we developed in this thesis. The following section introduces the JACK agent system, and shows how it follows the BDI model of agents.

2.4 JACK Intelligent Agents

The agent development platform that we used as a testbed for evaluating the reasoning algorithms presented in this thesis, is the JACK Intelligent Agents^{TM} system (Busetta et al. 1998). JACK, from Agent Oriented Software[5] is a state of the art agent development toolkit for building intelligent agents as components of Java applications. JACK provides the core architecture and capability for developing and deploying software agents in distributed applications (Busetta, Rönnquist, Hodgson & Lucas 1999). It has been successfully used to build a variety of large scale and significant applications.

Of the many systems that have implemented the BDI model, two are of particular relevance to JACK, since they represent its immediate predecessors. The first

[4]Note that internal events get added into the event-queue as they occur.

[5]www.agent-software.com

generation is the Procedural Reasoning System (PRS) (Ingrand et al. 1992) developed by SRI International in the mid 1980's. dMARS (d'Inverno et al. 1998), built in the mid 1990's by AAII[6] is the second-generation system. dMARS has been used as the development platform for a number of applications, including simulations of tactical decision-making in air operations and air traffic management. dMARS was also used by NASA in space shuttle monitoring applications.

The JACK agent language is an extension to Java, and hence it inherits all the advantages provided to the programmer by Java, such as being type-safe and the portability of code. It also guarantees a widely available, well-supported execution environment. The Java based modular approach, and JACK's own careful modular design, allows JACK to be extended without the necessity of having to fully understand and change the internal low level implementation of JACK. This makes JACK well suited to experimentation with new agent architectures, such as implementing and evaluating the reasoning framework presented in this thesis.

JACK agents (i.e. agents built using the JACK framework) are based on the BDI model of agents, and follow an execution cycle that is similar to that described in the previous section.

Like many implementations of the BDI model, a goal in JACK is represented as a type of event which is transient in that if it cannot be achieved in the current state of the agent, the goal is discarded and not stored for future reference.

There is a *plan library*, which is a pre-defined set of plans where each plan handles a particular event. A plan consists of a *context condition*, which determines its applicability in the current situation of the agent, and a plan body. The plan body may contain a combination of sub-goals, belief updates or arbitrary Java code. Sub-goals are handled in the same way as top-level goals (i.e. via events and plans). In order for a plan to succeed, all of the instructions (including sub-goals) within the plan must successfully complete.

The general execution model of a JACK agent (Busetta et al. 1999) can be mapped to the abstract BDI interpreter of Rao and Georgeff (Rao & Georgeff 1995) and is outlined below and illustrated in figure 2.2.

A goal may (and usually does) have multiple plans that handle it. These are the *relevant plans* for that goal. When a goal is generated[7], the agent retrieves the relevant plans from the plan library for that goal, and selects the plans that are applicable in the current context of the agent, by evaluating the respective context conditions of the plans. The resulting set of plans is termed the *applicable plan set* of the agent.

[6]The Australian Artificial Intelligence Institute in Melbourne, Australia.

[7]The semantics of when an agent begins goal execution is not well defined, but in the simplest case it is when the agent takes on the task of satisfying the goal.

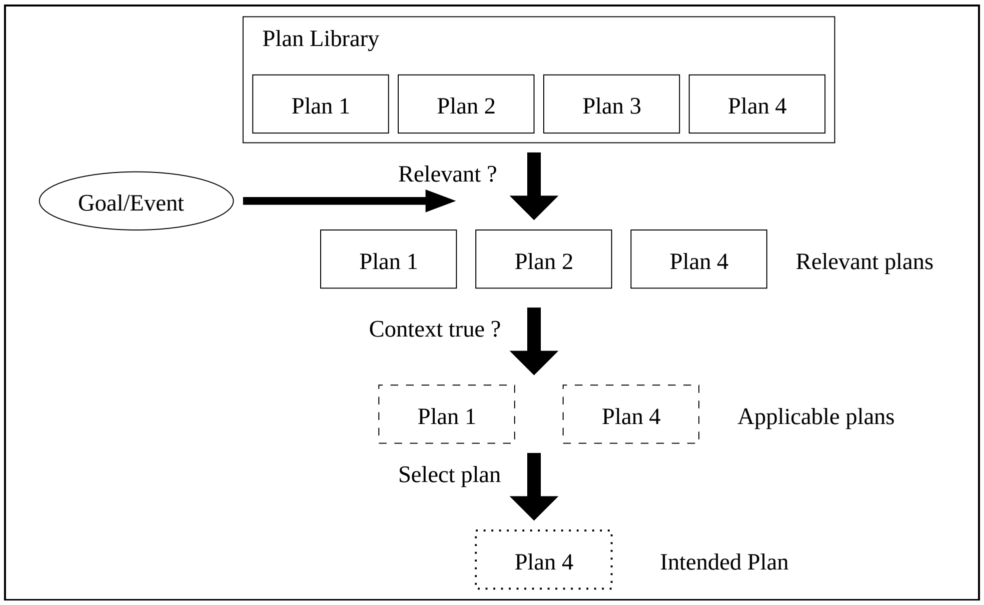

Figure 2.2: *Agent execution model: Plan selection.*

From this set of applicable plans, the agent selects one plan and begins execution. The choice of plan depends on the preference mechanism chosen. JACK has three mechanisms for constraining the order in which plans are tried: the use of meta-plans, a numerical priority assigned to each plan (precedence), and the order in which they are listed in the agent (prominence) (meta-plans override precedence which in turn overrides prominence). If the selected plan fails then another applicable plan (that has not already been tried) is chosen and executed. This provides persistence in achieving a goal and robustness in changing environments.

If all applicable plans fail then the goal achievement fails. In the event that no applicable plan is available, unlike systems like JAM (Huber 1999) that maintain a record of the goal for future re-attempt, JACK discards the event.

Within the JACK framework there is currently no notion of checking for consistency, in terms of conflicts between goals and plans.

2.5 Planning Agents and Hierarchical Task Networks

While practical BDI systems lack generic reasoning mechanisms about goal interactions, planning systems address these issues using various techniques that are specific to the type of planning system. In particular, the work of Clement et al. (2001, 2001, 1999*a*, 1999*b*), some of which was done in parallel to this thesis, is relevant. Prior to discussing the work of Clement et al., in order to better understand their work and the differences between their approach and that of this thesis in managing conflicts, it is important to understand the differences between planning in general, and specifically between hierarchical task network (HTN) planners and BDI systems.

Planning in general involves formulating a course of actions to achieve some desired objective(s) (Russell & Norvig 1995, chapter 11). There are three categories of planning (Smith, Frank & Jónsson 2000): classical-planning, decision theoretic planning and hierarchical task networks.

Classical planning is the traditional STRIPS style planning (Fikes & Nilsson 1971), where given a set of actions, an initial state and a goal state to be achieved, actions are chosen and ordered such that they achieve the goal state. Actions are defined using pre and post conditions, which specify the state of the world that must be true for the action to begin, and the changes to the state after the action is performed, respectively. The states are specified as a set of conjunctive literals (recent work has been done on representing disjunctive literals as well (Pednault 1989)). Classical planning is useful in domains where it is possible to pre-determine a set of actions for an agent to follow. This is not sensible in the highly dynamic domains that BDI agents operate in.

Decision theoretic planning takes uncertainty into consideration (Blythe 1999). It involves a state space in which transitions between states are probabilistic in nature. For example, sending a message from agent A to B may have a 70% probability of success depending on the reliability of the communication link. If the link fails, then the message may need to be re-sent. Traditionally, the problem of mapping the state space into actions (called a policy) is solved using techniques called *value-iteration* (Bellman 1957) and *policy-iteration* (Howard 1960). These basically specify which action to perform in every possible state. The large state space causes these methods to have a high complexity.

Much of the practical work on AI planning systems in the recent years has been based on hierarchical task network (HTN) decomposition (Erol, Hendler & Nau 1994*b*, Erol, Hendler & Nau 1994*a*). HTN planners are the popular choice for industrial applications as they, much like BDI systems, allow the domain experts to encode any available domain knowledge about the hierarchy between tasks, and the inter-relationship among actions within a task. This information enables the planning system to be more efficient and to produce plans that are more expressive, which aids in plan related reasoning such as plan selection. RETSINA (Paolucci, Shehory, Sycara, Kalp & Pannu 2000), SHOP (Nau, Cao, Lotem & Mũnoz-Avila 1999) and DECAF (Graham, Decker & Mersic 2003) are examples of modern HTN planning systems.

HTN planners and BDI systems are similar in certain ways, but different in others. These similarities and differences are highlighted as HTN planners are described below.

HTN planning works by reducing high level abstract tasks (similar to goals in BDI) to low level primitive tasks (actions). This is achieved by recursively expanding the high level tasks into lower level task networks. Each task network may consist of primitive tasks and/or abstract tasks that require further decomposition. Decomposition methods for each abstract task are pre-defined by the programmer and there may be more than one decomposition method for a given abstract task, much like the plans in the plan library of a BDI system. Figure 2.3 illustrates this basic procedure of HTN planning[8].

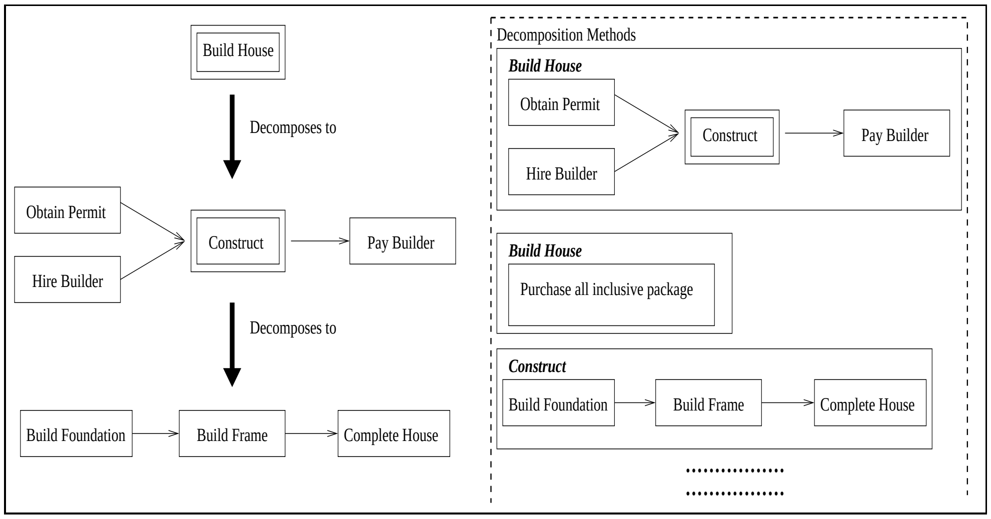

Figure 2.3: *Hierarchical task decomposition.*

The main advantage of HTN planners over classical planning is that the search for solutions is more tightly controlled since HTN planners are told how to use actions via the decomposition methods, while classical planners must figure this out from the action descriptions. The disadvantage of having pre-specified decomposition methods is that it is not always possible to specify a comprehensive set of decomposition methods for an application. Another drawback is that the planner cannot accomplish goals for which a decomposition method is not specified.

In BDI systems, the ordering of actions and sub-goals are built into a plan at a design level. These orderings are followed at run-time when plans are executed, and cannot be modified. In HTN planners as well, ordering constraints are placed

[8]Note that in the left hand diagram in fi gure 2.3, each level illustrates the decomposition of an abstract task and does not display the complete plan.

at a design level in the decomposition methods of abstract tasks. However, unlike BDI systems, it is possible to modify these constraints at run-time (either by removing them, or adding new constraints). Also, since HTN planners find complete solutions, it is possible to place ordering constraints across plans. This feature is useful, especially for methods that resolve conflicts in the resulting task networks after decomposition.

Current BDI systems allow conflicting goals to be pursued simultaneously. HTN planners on the other hand, ensure that decomposed tasks are consistent with other tasks within the system by the use of *critics*. A *critic* is a method that takes a task decomposition with potential conflicts and outputs a solution that is correct with respect to the purpose of the critic. For example, a critic that resolves variable binding constraints outputs a solution with correct variable bindings in the solution. Another critic that resolves resource conflicts, outputs a solution that is conflict free with respect to resources. Whilst critics are useful, they are computationally costly, depending on the branching factor of the task hierarchy and the number of interacting conditions in each task in the network (Erol et al. 1994*b*).

The basic procedure of HTN planning is to iteratively decompose abstract tasks, checking for conflicting situations and resolving them. Conflict resolution is achieved by means of imposing additional ordering constraints, and combining or eliminating overlapping actions, which is the same procedure to that used in classical planning. The planning process will be complete when the resulting network only consists of primitive tasks that, together with the constraints placed on them, are consistent in terms of what they achieve. These solutions are linearised, which does not allow for future uncertainty in the domain. Like classical planning, this procedure completes, yielding a sequence of actions, before any actions are performed.

Due to the dynamic nature of the environments in which BDI agents often operate, it is not sensible to find fully defined solutions at the start of execution. The BDI approach of choosing plans only when a sub-goal is ready to execute, allows the agent to adjust to the current state of the environment.

An HTN planner could also allow for uncertainty in two ways. The first, is if it does not attempt to find a complete solution and decomposes abstract tasks to partial levels. For example, it could decompose a task to a single level, and delay the decomposition of any abstract tasks in the resulting network, until they are actually needed. When they are required, they too can be decomposed to a single level and so on. However, this would mean that it would not be possible to find consistent solutions as future decompositions are unknown. This is precisely the way in which BDI agents operate. The second way of dealing with uncertainty in HTN planners, is for the planner to generate a solution, the agent to execute a step (or two) of the solution and re-call the planner to find a more up-to-date solution in the agent's new context, execute a step (or two) in the new solution and so on. This however, would incur a heavy overhead as the algorithms for finding consistent solutions are complex.

A further difference between HTN planners and BDI systems is that in HTN planners tasks are added into the system at the same time (i.e. in batches) and if a new task is to be pursued, then a re-plan of all existing tasks is required. In contrast, BDI systems handle each goal individually, and goals may be added into the system at any time. One of the main reasons that HTN planners require a re-plan every time a new task is to be added is to ensure that the decompositions of tasks are consistent with each other. The algorithms that we develop in this thesis manage goal interactions, maintaining the flexibility of being able to add goals into the system at any time. This is done without the need for a complete re-assessment of the currently executing plan(s).

2.6 Conflict Management

There are two aspects to managing conflicts between goals: conflict detection and conflict resolution (or prevention in the case where the conflict is yet to occur). This section investigates existing work in both these aspects of conflict management in the area of multi-agent systems, traditional planning and HTN planning.

Multi-Agent Systems

There has been a lot of work on managing conflicts in a multi-agent setting (Bond & Gasser 1988, Tessier et al. 2000). In a multi-agent setting where multiple agents compete for the same resources, conflicts with respect to resources are common (Tessier et al. 2000). These agents may also act simultaneously within the same environment and these actions could interfere with each other, causing conflict (Boutilier & Brafman 1997). In such environments, agents do not have direct access to the mental attitudes (i.e. beliefs, plans, goals etc.) of other agents and conflict detection and resolution can be performed only on the limited knowledge that is shared or published by each agent.

Methods for conflict detection in a multi-agent setting (Tessier et al. 2000) are not precisely defined. In general, intelligent agents do not plan ahead and therefore conflicts are not foreseen, but are detected as they arise. For example, in the case of resource bounded agents, when an agent requests a resource that is already in use, or when two agents simultaneously request access to the same resource, a conflict can be detected.

If resource allocation is not monitored, and agents access resources concurrently, the agents may fail in their goal pursuit due to resource related conflict. For example, if two agents draw power from a central, shared energy source that has 110 units of energy available, and they each begin pursuit of a goal that requires 100 units of energy, clearly either one or both goals will result in failure due to the lack of resources. In this case, a conflict is detected when the goals fail, which is too late and is best prevented if possible.

Situations of greater concern occur when conflicts go undetected and do not result in failure or deadlock, but in infinite cycles (i.e. live-lock (Kobayashi 2000)). One such example is illustrated below in figure 2.4.

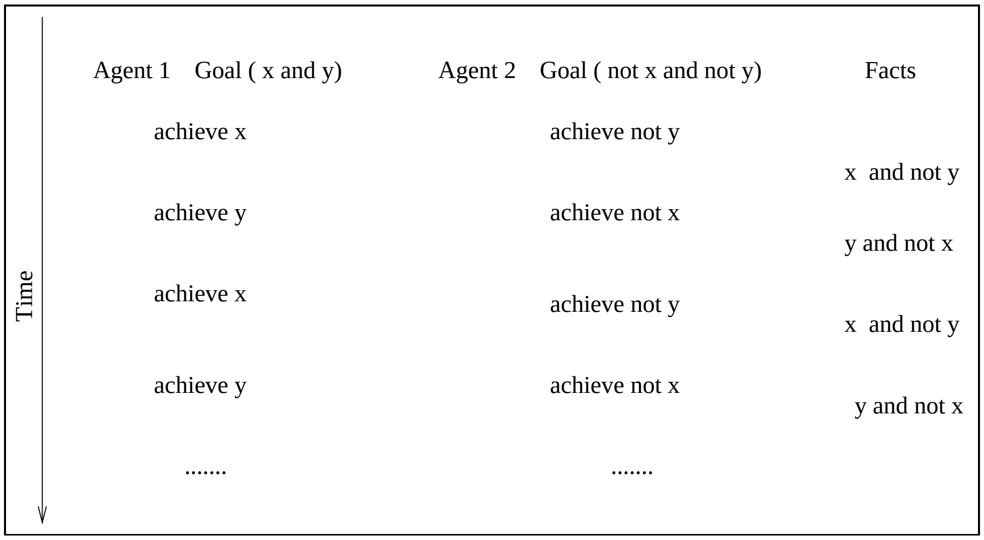

Figure 2.4: *Two agents pursuing conflicting goals, caught in a live-lock situation.*

In practical BDI systems like JACK (Busetta et al. 1998), after *Agent1* achieves *x* and *y*, it will not notice when they are undone by the other agent (unless explicit checking is programmed into the application). In this case, the agents do not enter an infinite cycle, but the final state will be *y and not x*. Therefore, neither of the agents will have achieved their goals.

The main focus of the theory on conflict management in multi-agent systems, is on conflict *resolution* in the event that two or more agents compete for the same resource or task allocation. Conflict resolution techniques (Bond & Gasser 1988) include argumentation (Jennings, Parsons, Noriega & Sierra 1998, Jung, Tambe & Kulkarni 2001), negotiation (Sycara 1988), voting (Ephrati & Rosenschein 1991), constraint relaxation (Sathi & Fox 1989), arbitration (Steeb, Cammarata, Hayes-

Roth, Thorndyke & Wesson 1988), self-modification (Sugie, Inagaki, Ono, Aisu & Unemi 1995), priority conventions (Ioannidis & Sellis 1989) and standardisation and social law (Shoham & Tennenholtz 1994).

The most common way of resolving conflicts between agents that communicate is via negotiation (Sycara 1988). The basic concept is for an agent to develop a proposal to resolve any conflicts that may arise and for other agents to generate counter proposals. This iterative process continues until an agreement is established between the agents involved in the conflict. The negotiation process is generally flexible and may include other conflict resolution methods such as argumentation (Jennings et al. 1998). Argumentation based negotiation follows a similar protocol to negotiation using proposals and counter-proposals, with the addition that these proposals are accompanied by explicit justifications (or arguments). Arbitration (Steeb et al. 1988) is a useful process of resolving conflicts when two agents negotiating to resolve a conflict get into a deadlock situation. In this situation, an arbitrator agent reviews the negotiation dialogues and generates alternative proposals, possibly involving constraint relaxation techniques (Sathi & Fox 1989), setting time limits or other techniques.

Among techniques used to resolve conflicts that do not involve negotiation, Ephrati and Rosenschein (1991) suggest an economic decision process that can be used to obtain multi-agent consensus, namely, the Clarke tax mechanism (Clarke 1971). Consensus is reached through the process of voting; each agent expresses its preferences, and a group choice mechanism is used to select the result. Shoham and Tennenholtz (1994) and Conte and Castelfranchi (1995) discuss methods of using social norms and standards that each agent should adhere to in a multi-agent system. This could potentially avoid conflicts.

Social norms assist in managing conflicts between agents. However, they are not relevant in managing conflicts internal to an agent, except in situations where goals affect each other, resulting in social norms being violated, which is beyond the scope of this thesis.

All of the above techniques, with the exception of that involving social norms, require the agent to communicate its preferences to other agents. This gives rise to an emphasis on trust (Marsh 1994) between agents and the correctness of the information communicated. The information communicated is also limited, as in most practical applications it is not feasible to communicate large amounts of data, due to bandwidth restrictions and latency in communication.

In contrast, if the conflicts are within a single agent, the agent has direct access to its mental states at any point in time, to obtain *complete*, *up to date*, and *accurate* information. The agent may also revise its mental attitudes, making conflict resolution within an agent simpler in comparison to the complex, sophisticated mechanisms required to resolve conflicts between agents.

Conflicts between agents are usually detected as they occur. Whilst this is acceptable, as the agents are not always aware of each others future actions, intelligent agents should recognize conflicts within themselves prior to their occurrence.

Traditional Planning Systems

Systems that use traditional planning (e.g. classical planning), typically have two stages: the planning phase and the execution phase. The planning phase generates the plans and the execution phase carries out these plans. In these systems, conflicts are managed in two ways:

- conflicts are detected and resolved at the planning phase (Russell & Norvig 1995, page 337-366); or
- the agent is allowed to execute the plans irrespective of whether they are inconsistent[9], and if they fail, re-plan a new solution in the new state of the agent (Russell & Norvig 1995, page 401-403).

The first approach ensures that the solutions are consistent, but can be computationally expensive, as the planner searches through the entire state space. The latter, is more efficient in terms of computational cost but could lead to problems, as some conflicts may not always lead to failure and for some, detection of conflict after failure may be too late, as discussed above in the case for multi-agent systems.

One of the key differences between planners and BDI systems is that planners plan ahead. They find solutions that consist of a number of immediate and future steps for the agent to follow. This allows the agent to reason about conflicts at an early stage as the future actions are known. The drawback is that it does not allow for uncertainty in the domain.

In terms of detecting conflicts, there are two main types of conflicting interactions that the planner seeks to uncover:

- The first are resource conflicts. In order to reason about resource conflicts, it is necessary to extend the pre-conditions and effects to include information about the resource needs of the action (in the pre-conditions) and the resources consumed or produced by the action (in the effects) (Russell & Norvig 1995, page 386-389). It is possible to check for resource conflicts by simply keeping track of the minimum and maximum amount of resource needs at each plan step, and then at the planning stage, ensure that the maximum resource needs of the complete plan are satisfied. There are other

[9]A consistent solution is where the ordering of actions do not contradict each other and the effects of one action do not undo established pre-conditions of another.

more sophisticated techniques of managing resource conflicts in traditional planners, such as the use of resource time maps (Koehler 1998).

- The second, more common form of conflicting interactions in classical planning, is when an effect of one action invalidates an already established condition that is a pre-requisite (pre-condition) to another planned action. Two such examples are illustrated in figure 2.5. In case a), the dependency link between the effect of the step *Go Home* and the pre-condition of the step *Watch TV* is termed a *causal-link* (Russell & Norvig 1995, page 346-349). A conflict occurs if this causal-link is undone.

  In case b), the agent has bound the same value (i.e. *car*) for the variable in both goals, which creates conflict.

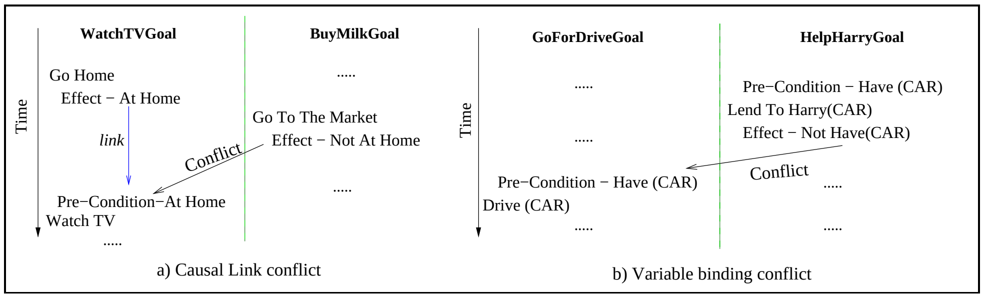

Figure 2.5: *Conflicts in STRIPS style planners.*

Conflicting interactions can be identified by the following process (Russell & Norvig 1995, page 349-358). When a solution for a plan is generated, data structures are required to record causal-links as steps are added to the solution. Before a new step is added into the solution the planner checks whether the effects of this new step introduce either:

(a) effects that invalidate already established causal links within the solution, or

(b) causal links that may be violated by the effects of actions that are part of the current solution.

The techniques that identify these interactions vary in different planning systems, from performing brute force state space search, to using heuristics and data structures to guide and reduce the complexity of the search (Weld 1999, Blythe & Veloso 1992).

In this work, we also consider protecting causal links, as well as protecting conditions that must be maintained during the execution of a particular goal or

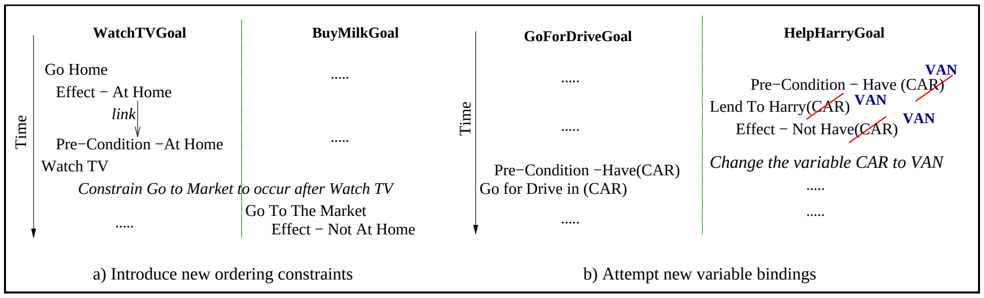

Figure 2.6: *Resolving conflicts by changing temporal constraints/variable bindings.*

goal step. For example, a goal to re-fuel may require the agent to remain stationary until re-fuelling is complete. The data structures and algorithms to detect conflicts that we develop in this thesis are more efficient than those in traditional planning, mainly due to the fact that we do not attempt to find consistent solutions to a problem, as done in planners. Our algorithms identify potential conflicts at run-time and guide the execution of plans so that these conflicts are avoided.

In planners, the actions are pre-determined and hence are known to the agent, which makes it possible to order them prior to execution, to avoid conflicts. In BDI systems, the agent typically has multiple ways of achieving a particular goal and its sub-goals. The choice of plans which it commits to, are made at run-time, as determined by the current situation that the agent is in. Our algorithms manage conflicts dynamically as the agent makes these decisions at run-time. While the agent is aware of the possible plan choices to achieve a particular goal, it cannot pre-determine the future choices. The search for interactions between goals therefore involves not just finding interactions, but also classifying them into definite conflicts which will occur no matter which way the agent goes about achieving its goal, and potential conflicts, which while not definite, could potentially occur depending on the choice of plans. These classifications are important to guide the agent's future decisions.

Conflict resolution in traditional planners (Fikes & Nilsson 1990) is done as follows:

During the search for the correct sequence of operators if a conflict is detected, the planner first attempts to place ordering constraints or different variable bindings on the conflicting operators to remove the conflict. Figure 2.6 illustrates such an example, which resolves the conflicts identified in figure 2.5.

If this is not possible, the planner backtracks to the last step which had no conflict, and attempts to search for different operators to achieve the goal state that

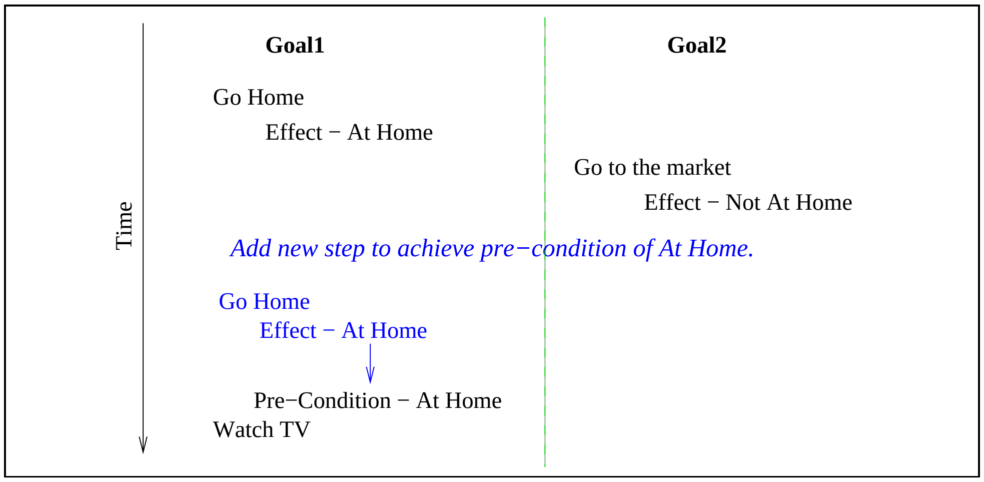

Figure 2.7: *Resolving conflicts by adding new plan steps.*

does not cause any conflict.

Another option in such situations of conflict, that early planners such as Warplan (Warren 1976) followed, is to introduce new steps to achieve the undone preconditions as shown above in figure 2.7. This approach however can introduce redundant steps as well as result in infinite cycles.

Apart from being complex, traditional planners do not allow for concurrent actions to be represented. This issue was addressed by Boutilier and Brafman in their work on planning with concurrent interacting actions (Boutilier & Brafman 1997). They provide a representation of concurrently interacting actions by extending the STRIPS action representation language. They add a *concurrent action list* to the STRIPS action descriptions, which specifies the actions that may and must not be executed in parallel to that action. This addition is simple yet effective in its purpose. They show this by modifying an existing STRIPS style planner, POP (Weld 1994), to include their representation for concurrent actions, generating nonlinear sequences of plans.

BDI agents do not perform any planning by combining actions to form plans that satisfy a goal. In BDI systems, plans are already pre-defined. These plans are chosen at run-time and are partial; they are further decomposed as the agent executes over time. Whilst not similar to classical planning, BDI systems are similar to Hierarchical Task Network (HTN) planners as described in section 2.5. We will now look at conflict management in HTN planners.

Hierarchical Task Networks

Hierarchical Task Network (HTN) planners use *critics* to ensure consistency between decomposed tasks (as discussed in section 2.5). A *critic* is a piece of code that takes a proposed, possibly inconsistent solution as input and outputs a consistent solution with respect to the critic's functionality. HTN planners use many kinds of critics including variable binding critics, redundancy critics and temporal constraints critics.

Variable binding critics are much like constraint satisfaction problem (CSP) solvers. That is, given a set of variables, a domain of values that can be assigned to each variable and a set of constraints on the values that groups of variables can take, these critics attempt to assign values to variables without violating any constraints. Redundancy critics attempt to identify and merge two or more steps that achieve the same pre-condition, so long as no constraint is violated. Temporal constraint critics, when possible, find a minimum set of ordering constraints on tasks that avoid interfering interactions and resource conflicts. These critics operate much like the algorithms for detecting and resolving conflicts in STRIPS style planners, with the exception that they have hard-coded domain knowledge and their search space is smaller due to the nature of HTN planners. This allows them to be more efficient than traditional planning techniques. However, their complexity in the worst case is still exponential (Erol et al. 1994*b*) given that they perform a search which can be exhaustive.

A key difference between HTN planners and the type of BDI agents that we focus on, is that BDI agents do not plan ahead. HTN planners look ahead and find a series of decompositions that lead to a consistent set of actions. BDI agents do not plan ahead, and it is often not practical to do so, given the dynamic nature of the environment.

HTN planners that find linearised solutions do not allow for uncertainty, which is not favourable in dynamic domains. The work of Clement et al. (1999*c*, 2001, 2001, 1999*a*, 1999*b*), address this issue in HTN planners by attempting to find non-conflicting solutions at abstract levels where possible. This allows for decomposition at a later, more informed, stage. Their approach is the closest related work to this thesis.

The work of Clement et al. uses the notion of *summary information* similar to the way in which we use it in this thesis. While the concept of summary information is common to both their work and ours, the data structures and algorithms differ as their work is designed for HTN planners and we focus on BDI agents. Their work on finding solutions at abstract levels is significant for HTN planners as it allows for a certain level of uncertainty, making HTN planners more suitable for dynamic domains. Their evaluation showed that using summary information to prune the hierarchy (i.e. collapse branches) at a higher level is useful, and it reduced the complexity greatly. However, when pruning was not possible and solutions only existed at the lowest levels, then the overhead of maintaining summary information was quite significant, with the complexity growing exponentially as the branching factor and the number of summary conditions increased (Clement & Durfee 2001, page 211-212).

The methods we present in this thesis do not suffer from such high complexities. This is mainly due to the fact that we do not attempt to find consistent solutions, but simply identify conflicting situations, and perform online monitoring and scheduling to avoid them. At an implementation level, Clement et al. attach resource summaries to actions and propagate them upwards to the level of abstract plans, while we attach summary information to goals and plans, with no distinction between abstract and concrete plans. The algorithms they present (Clement et al. 2001) allow for expansion and scheduling of appropriate plans prior to execution and need to be re-run if re-planning is required at a later stage. In contrast, our algorithms do some compile-time reasoning and then at run-time use more efficient algorithms to dynamically update the data structures and reason about conflicts.

A valuable advantage of the approach that we present is that goals may be added at any time, and the algorithms will dynamically manage the interactions at run-time. However, in planning approaches, including that of Clement et al., if new tasks were to be added into the system, a re-plan would be necessary, which would create a significant overhead.

In general, all the work in the planning domain is applicable only when it is possible to plan ahead. In BDI agent systems however, the plan to be used is chosen at runtime, based on the current context of the agent. This approach is important, because, in general, agents operate in dynamic environments, where things outside the agent's control are likely to change the environment and thus the possible ways to achieve a goal that the agent is pursuing.

2.7 Positive Interactions

As in the case of conflict management, there is much work on co-operation between agents but little work on positive interactions between multiple goals within a single agent.

Teamwork involves a group of agents acting together to achieve a common goal. The *Joint-Intentions* framework of Cohen et al. (1997), the model of *SHARED-PLAN* by Grosz and Sidner (1990), the *STEAM* model of team work by Tambe (1997), which incorporates both *Joint-Intentions* as well *SHAREDPLANS*, and the *SOAR* architecture of Newell (1994, 1992) all address the issue of working together as a team. Teamwork is not the kind of positive interaction that is addressed in this thesis as it deals with a single agent that has its own individual goals.

One type of positive interaction that could occur within a single agent is where two different plan instances of two goals of an agent can be merged, resulting in just one of the plan instances being executed to satisfy the requirements of both respective goals. This is termed *plan merging*.

Similar to goal conflicts, there is much work on managing the process of plan merging in systems where the plans to be executed are determined before execution (Sacerdoti 1977, Wilkins 1988, Tate 1977, Horty & Pollack 2001, Foulser, Li & Yang 1992, Nau, Yang & Hendler 1990, Cox & Durfee 2003). Although they are not directly applicable to BDI systems, they do have conceptual value that is used in some of the algorithms developed in this thesis. Some of this related work is outlined below.

In the planning domain, Sacerdoti's NOAH (Sacerdoti 1977) was one of the first planners to allow plan merging. Similarly to managing conflicts, it used *critics* to identify possible plan merges and to eliminate redundant operators and/or impose ordering constraints to facilitate the merging of plans. Redundant operators are detected if there is more than one operator in the current plan that achieves the same pre-condition. Other planners such as Wilkin's SIPE (Wilkins 1988) and Tate's NONLIN (Tate 1977) have more advanced capabilities such as the ability to recognize if a pre-condition of some operator can be achieved by a side-effect of another operator in the current plan. In such a situation, the original operator that was meant to achieve the pre-condition is replaced by a dummy operator and ordered to occur along with the operator whose side-effect achieves the same pre-condition. This process is termed *goal or operator phantomization* (Young, Pollack & Moore 1994).

Foulser et al. provide a formal theory for plan merging and present both optimal and efficient heuristic algorithms for finding minimum-cost merged plans (Foulser et al. 1992). They discuss the conditions under which plans may be merged and the relative complexities. Their work is based on the STRIPS operator definitions (Fikes & Nilsson 1990) and has served as a foundation for other more recent work on plan merging (Cox & Durfee 2003). The algorithms they propose to detect and to facilitate plan merging, rely on the STRIPS style of planning-ahead, where it is assumed that the devised solution will definitely be executed. Therefore, the algorithms are not suitable for BDI systems. Their work makes use of a cost that is assumed to be associated with each operator, in order to determine the minimum-cost merged plan set. In this thesis, the plans do not represent cost and the algorithms on plan merging do not attempt to find an optimal solution when multiple options exist. However, a mechanism for choosing a particular merge option could be incorporated into the algorithms and could be based on cost or some other form of utility depending upon the application requirements, but is not pursued in this thesis.

Horty and Pollack (2001) build on the work of Foulser et al. (1992) and present a theory of rational choice, where an agent evaluates its options in the context of its existing plans. Their work is interesting in that they evaluate potential goals in the context of other goals being pursued. For example, assume an agent is pursuing a goal of buying a shirt and considering a new goal of buying a tie. In this case, even if buying a tie is not so important, since buying a tie and buying a shirt have many aspects in common (such as going to the shopping centre, going to a clothing shop etc.), the agent may consider purchasing a tie, as its additional cost would be much less if pursued together with buying a shirt.

In their work, Horty and Pollack discuss ways in which plan steps of similar type can be merged. In order to determine if combining two plans is beneficial, they present mechanisms for calculating the cost of the plans in isolation and for calculating the cost of the combined plans (which is done in a similar fashion). If the cost of the combined solution is less than the cumulative cost of each plan in isolation, then there is a benefit. An important thing to note here is that while it seems intuitive that combining plans always leads to a cost benefit, this is not always true. For example, in the previous scenario of buying a tie and a shirt, if the cost of the tie is extremely expensive at the shopping centre where the shirt is purchased, then despite the common task of going to the shopping centre, purchasing the tie may no longer be beneficial. The work of Horty and Pollack does, like many others, require a complete plan ahead with explicit representations of ordering constraints and causal links, where a *causal link* is the dependency relation between two plans where the first plan achieves an effect that is a pre-condition to the other. They also assume that plan steps are atomic and cannot concurrently occur, while we allow plan steps to be executed in parallel.

In earlier work, Pollack (1991) provides a discussion of how the current plans of an agent can constrain the reasoning about new goals. For example, if an agent decides to pursue a new goal of buying bananas and already has a current plan (intention) to go to the supermarket, it is common-sense reasoning for the agent to use this current plan to satisfy the new goal. This process is termed *plan overloading*.

Pollack also argues that the agent, after further deliberation about all possible alternate ways of achieving its new goal, may realise that the overloading option may not be the most optimal means of achieving the goal. Although a sub-optimal solution, the cost of reasoning about the optimal solution (which can be exponential) may not be justified when overloading is simple, common-sense reasoning. Pollack does not however provide detailed mechanisms on how plans (intentions) may be overloaded, but the theory certainly motivates the work presented in this thesis on managing the positive interactions between goals.

The work of Cox and Durfee (2003) on discovering and exploiting synergy between hierarchical planning agents addresses reasoning about positive goal interactions in hierarchical task networks. Their work was done in parallel with the work presented in this thesis, and builds on the work of Clement and Durfee (1999*b*) that used the notion of summary information to co-ordinate concurrent hierarchical plans. Cox and Durfee attempt to find overlapping effects between plans in hierarchical task networks, allowing solutions to be found at abstract levels when possible. This allows for redundant plans to be dropped improving the overall efficiency of the system. They use the notion of summary information to prune the search space when looking for overlapping effects. This is done by pruning branches whose summary information does not contain any overlapping effects (or post-conditions, as they term it). Where overlapping effects exist, further refinement (search) is required to identify the plans that can be merged and to place explicit orderings to facilitate the merge. This is possible because the solution is a complete plan.

In comparison, the approach presented in this thesis is suited to BDI systems which do not plan ahead due to the dynamic nature of the environment. Also, in these systems, goals are added at different times into the system and all goals are not known to the agent when it begins. For these reasons, our algorithms do not attempt to place any off-line ordering constraints. Our algorithms do however, recognize situations where plans could be merged and, if such situations arise, perform online scheduling to facilitate the merge process. The data structures we develop to manage positive interactions that contain all the necessary information to reason about plan merges, are maintained at abstract levels. Therefore, a refinement search is not required. In addition, Cox and Durfee's approach supports only pairwise merges in a single discovery and multiple plan merges can be done only through repeated discovery and adoption of pairwise merges, which is costly. On the other hand, our algorithms are not restricted to pairwise plan merges, and support multiple plan merges.

In contrast to the related work described above, the algorithms we present in this thesis identify positive interactions between goals without requiring each goal to have a fully elaborated plan that achieves it. Plan merging is done through a limited form of scheduling. For example, suppose that the goals G_1 and G_2 have a common plan P that could potentially be performed only once for both goals rather than once for each goal. The goal execution is then synchronised so that P of G_1 occurs simultaneously with P of G_2, and when the situation arises only one instance of P is executed. The synchronisation of goal execution is done by monitoring the plan execution for when P, of either G_1 or G_2 is reached, and suspending that plan until the P of the other goal is reached.

While most planning solutions perform off-line scheduling, do not consider plan failure, and include explicit representations of time, the algorithms we develop in this thesis perform online scheduling and monitoring of plans taking plan failure into consideration. Although time constraints are not explicitly represented, sufficient information is generated to perform reasoning about future positive interactions. As with the mechanisms that manage conflicts, there is a strong emphasis on providing detailed mechanisms for managing positive interactions that can be directly implemented in agent implementation platforms such as PRS (Ingrand et al. 1992), JAM (Huber 1999), dMARS (d'Inverno et al. 1998), and JACK (Busetta et al. 1998).

Chapter 3

Goals : Representation and Reasoning[†]

In this chapter we develop data structures to represent goals and plans that capture the desirable properties of goals and that allow us to reason about goal interactions. Section 3.1 present our structures, and section 3.2 describe how they realise the desirable properties of goals. An important feature of these structures is their ability to be easily implemented in practical agent systems. This is highlighted in the prototype system developed as part of this thesis to evaluate the algorithms for managing goal interactions.

The relationship between goals and plans as defined in BDI systems lead to a a natural tree like hierarchy. This hierarchy is implicit within BDI systems. We represent it explicitly and exploit its advantages in the algorithms that we have developed to manage goal interactions. In section 3.3 we present this structure, which is termed a *goal-plan tree*. We also show how the tree can be useful in the algorithms that are developed.

The attributes of goals and plans that we represent are not new to the area of agents and originate from various sources, including work in planning. However, there is no single work that represents goals and plans with the collection of attributes that are required for our approach to reason about goal interactions.

In the last section, section 3.4, we discuss in detail the kinds of reasoning about goal interactions that are addressed in this thesis.

[†]Part of the work presented in this chapter has been previously published (Thangarajah, Padgham & Harland 2002, Winikoff et al. 2002).

3.1 Representing Goals and Plans

In order to reason about goal interactions we define explicit structures for goals and plans. The structure for plans is important because plans play a key role in the reasoning process, since goals are achieved via plans. In the same manner in which a class is defined as a template for an object in an object-oriented programming language (Rumbaugh, Blaha, Premerlani, Eddy & Lorensen 1991, Booch 1994), a *goal-type* is introduced as a template for a goal. When an agent chooses to pursue a particular goal, a *goal-instance* of the goal-type is created (similar to how an object is instantiated from a class, in object-oriented terms). Similarly, a *plan-type* is defined as a template for a plan, and a *plan-instance* is created when the agent executes a particular plan-type.

Below is the Bachus-Naur Form (BNF) of the goal-type (GoalType) and plan-type (PlanType) structures as used in this thesis, followed by a discussion of their attributes.[1]

$\langle$GoalType$\rangle \rightarrow \langle$GoalTypeName$\rangle$ $\mathbf{\{}\langle$Plans$\rangle\mathbf{\}}$ $\mathbf{\{}\langle$In-Conditions$\rangle\mathbf{\}}$ $\mathbf{\{}\langle$Effects$\rangle\mathbf{\}}$
$\langle$GoalTypeName$\rangle \rightarrow \langle$Label$\rangle$
$\langle$Plans$\rangle \rightarrow \langle$PlanTypeName$\rangle$ ($\mathbf{,}$ $\langle$PlanTypeName$\rangle)^*$
$\langle$PlanTypeName$\rangle \rightarrow \langle$Label$\rangle$
$\langle$In-Conditions$\rangle \rightarrow \boldsymbol{\lambda}$ | $\langle$Condition$\rangle$ ($\mathbf{,}$ $\langle$Condition$\rangle)^*$
$\langle$Effects$\rangle \rightarrow \boldsymbol{\lambda}$ | $\langle$Condition$\rangle$ ($\mathbf{,}$ $\langle$Condition$\rangle)^*$
$\langle$Condition$\rangle \rightarrow \langle$Statement$\rangle$ | **NOT** $\langle$Statement$\rangle$
$\langle$Statement$\rangle \rightarrow$ **string** | $\langle$Variable$\rangle$ **=** $\langle$Value$\rangle$

$\langle$PlanType$\rangle \rightarrow \langle$PlanTypeName$\rangle$ $\mathbf{\{}\langle$Pre-Conditions$\rangle\mathbf{\}}$ $\mathbf{\{}\langle$In-Conditions$\rangle\mathbf{\}}$ $\mathbf{\{}\langle$Effects$\rangle\mathbf{\}}$
$\mathbf{\{}\langle$PlanBody$\rangle\mathbf{\}}$ $\mathbf{\{}\langle$Resources$\rangle\mathbf{\}}$
$\langle$Pre-Conditions$\rangle \rightarrow \boldsymbol{\lambda}$ | $\langle$Condition$\rangle$ ($\mathbf{,}$ $\langle$Condition$\rangle)^*$
$\langle$PlanBody$\rangle \rightarrow \langle$Action$\rangle$ | $\langle$GoalTypeName$\rangle$ | ($\langle$PlanBody$\rangle$ **;** $\langle$PlanBody$\rangle$) |
($\langle$PlanBody$\rangle$ || $\langle$PlanBody$\rangle$)
$\langle$Resources$\rangle \rightarrow \boldsymbol{\lambda}$ | $\langle$Resource$\rangle$ ($\mathbf{,}$ $\langle$Resource$\rangle)^*$
$\langle$Resource$\rangle \rightarrow \langle$ResourceName$\rangle$ $\langle$Value$\rangle$
$\langle$ResourceName$\rangle \rightarrow \langle$Label$\rangle$

$\langle$GoalInstance$\rangle \rightarrow \langle$InstanceName$\rangle$ $\langle$GoalType$\rangle$
$\langle$PlanInstance$\rangle \rightarrow \langle$InstanceName$\rangle$ $\langle$PlanType$\rangle$
$\langle$InstanceName$\rangle \rightarrow \langle$Label$\rangle$

[1]Note that the goal, plan and condition representations should be viewed as sentence schemas, that is, they may have arguments, but there is no quantifi cation over variables.

$\langle$Variable$\rangle \rightarrow$ **unique string**
$\langle$Value$\rangle \rightarrow$ **string**
$\langle$Label$\rangle \rightarrow$ **unique string**
$\langle$Action$\rangle \rightarrow$ **code defined by the programmer**

- **GoalTypeName** and **PlanTypeName -** A label which indicates the type of the goal or plan respectively.
  e.g. *GoToBeachGoal*, *GoToBeachPlan*.

- **Plans -** The set of plan-types that could possibly satisfy the goal. An agent could have more than one plan that achieves the same goal, but for every goal there *must* be at least one plan that could satisfy the goal. Therefore, this set of plan-types cannot be empty. Whilst having a plan does not necessarily imply that the goal pursuit will be successful, it states that the goal is *possible* under certain conditions as determined by the appropriate plan.
  e.g. *GoToBeachGoal.Plans(TaxiToBeachPlan, WalkToBeachPlan, ..)*.

- **In-Conditions -** The set of conditions that must hold during the pursuit of a goal/plan. If *any* of the in-conditions become false during the pursuit/execution of the goal/plan, the goal/plan is dropped with failure.
  e.g. *GoToBeachGoal.InConditions(WeatherIsSunny, not FeelingLazy)*.
  TaxiToBeachPlan.inConditions(HaveMoney).

- **Effects -** For a goal, this is the set of conditions that must be achieved in order for the goal to succeed. e.g. *GoToBeachGoal.Effects(AtBeach)*.

  For a plan, the effects are the set of conditions that are made true by the actions within the plan. This set does *not* include the effects of any sub-goals. This is because the programmer cannot pre-determine the run-time path that might be chosen to satisfy the sub-goals, which determines their effects. Attaching effects to plans rather than actions allows the programmer to write arbitrary code within the plan body, which is similar to that done in practical systems like JACK (Busetta et al. 1998). The onus is on the designer of the agent system to ensure that all relevant effects are identified for goals and plans. However, this is relatively straightforward as for a plan the designer only needs to include the effects of its actions and not the effects of its sub-goals. Similarly for a goal, only the desired end effects of the goal need to be included.
  e.g. *TaxiToBeachPlan.effects(AtBeach, OweTaxiDriver($20))*.

  The effects declared at each goal are the *intended effects* of that goal (also termed as *primary expectations* by Pollack (1991)[2]), and the effects of a plan that are not effects of the corresponding goal, are *side effects* of the goal (*secondary expectations*). For example, the plan to take a taxi to the beach may achieve the effects *AtBeach* and *OweTaxiDriver($20)*. By examining the effects of the goal (i.e. *AtBeach*) it can be determined that *OweTaxiDriver($20)* is a side-effect as it is not an intended effect of the goal.

- **Pre-Conditions -** Are specific to plans and are the set of conditions that must be true in order for an instance of the plan to begin execution. The conditions need *not* hold once the plan has begun execution.
  e.g. *TaxiToBeachPlan.preCondition(AtTaxiStand, HaveMoney)*.

  The pre-conditions are implicitly extended with the in-conditions. For example, the in-condition of *HaveMoney* is also included in the pre-conditions. If it is not included and *AtTaxiStand* is true but *HaveMoney* is false, the plan would begin, as the pre-condition is true, but fail immediately since the in-condition is not true.

- **PlanBody -** This represents the functionality of the plan and contains actions and sub-goals. These actions and sub-goals are combined by either sequencing them (e.g. "achieve goal G_1 and then perform action A", written as "$G_1; A$") or by performing them in parallel (e.g. "achieve goals G_1 and G_2", written as "$G_1 \| G_2$"). All actions are assumed to be atomic, while sub-goals are treated the same as top-level goals.
  e.g. *body*$(TaxiToBeachPlan)$ ::=
  $$HailTaxiAction; NegotiateFareGoal; PayFareGoal$$

- **Resources -** Are specific to plans, as they determine the resources required to achieve a goal. This attribute captures the resource requirements of all actions within the plan and does *not* include the resource requirements of any sub-goals of the plan.
  e.g. *TaxiToBeachPlan.Resources({(Money, 100)})*.

- **InstanceName -** This is an additional attribute that is attached to each goal and plan instance. The instance-name is a label that provides a *unique* handle to each goal/plan instance, as there can be multiple instances of the same goal/plan type. While all goal/plan type attributes are pre-assigned to each goal/plan type at compile-time, the instance name is assigned at run-time.

[2]Pollack defi ne expectations as beliefs about future activities and circumstances. *Primary expectations* are those that are directly intended by the agents actions and *secondary expectations* are those that are side effects of intended actions.

These structures collectively realise some of the desirable properties of goals which are described in the next section.

3.2 Desirable Characteristics of Goals

Section 2.1 described attributes of goals required by various researchers. The attributes realised by our representations are as follows:

Persistent: Once the agent begins pursuit of a goal, the goal will *only* be dropped if it succeeds or when there is a *good reason* for dropping it. We define a *good reason* to be one of the following:

- The agent realises that the goal state to be achieved has become true (by some other means other than its own plan execution). This could occur if an effect of a plan executed to satisfy another goal achieves the goal condition, or if another agent's actions satisfies the goal. Since we explicitly represent the effects of goals and plans, it is possible to monitor for such situations.
- The goal is in conflict with other goals and the preferred way of resolving the conflict is to drop the goal. In chapters 4 and 5 we present mechanisms that detail the process for managing conflicts.
- The agent has tried every possible way that it knows of achieving the goal, and failed. The possible ways of achieving the goal are represented by the set of plan-types attached to each goal.

When the agent pursues a goal, the agent first selects one particular plan to satisfy the goal, if this plan fails, the agent would then attempt to try another applicable plan (if available) to achieve the goal. This also brings about a level of *persistence* in the way that an agent satisfies its goals.

Unachieved (unfulfilled): The explicit effects of each goal allows an agent to realise when the goal has been achieved. If all the effects of a goal are true before the agent begins pursuing the goal, the goal is not pursued as it is already achieved. This ensures that the agent only pursues goals that are *unachieved*.

Possible (implementable, realisable): Each goal-type has an associated set of plans that achieve it; by definition, the set of plans cannot be empty. This makes the goal *possible* in at least one way, under situations that are determined by the plan(s). That is, it must have at least one plan that satisfies the goal.

While this does not guarantee that the goal will succeed, it ensures that the agent does not take on tasks that it has no way of achieving.

Consistent (non-conflicting, coherent): The algorithms developed and presented in chapters 4 and 5, assist in identifying and resolving conflicts between goals, making them *consistent* (i.e. conflict free, to the best knowledge of the agent). These algorithms utilise the pre-conditions, in-conditions, effects and resource requirements that are attached to each plan, and the effects and in-conditions that are attached to each goal.

Known: Each agent contains a list of active goals, and a list of goals that are suspended due to conflicts. This is made possible by the the explicit structure that we provide for goals and makes the goals *known* to the agent.

Discussion and related work

Pre-conditions and effects originate from early STRIPS style planning (Fikes & Nilsson 1990) where each operator is composed of the action description (which is analogous to the plan or goal type), pre-conditions and effects. Unlike in STRIPS planning where pre-conditions and effects are attached at the level of actions, in this thesis they are attached to plans. Pre-conditions are not attached to goals as they are specific to the plans that achieve the goal and not the goal itself.

The effects of a goal are realised by its plans. In addition to the effects of their relevant goal, plans also achieve other effects which we call *side-effects* (i.e. effects that are not direct effects of the goal). Therefore, at the level of plans we cannot distinguish between the effects that are direct effects of the goals from those that are side-effects. For this purpose it is necessary to explicitly state the (intended) effects of a goal.

With respect to effects, in STRIPS style planning every action brings about at least one effect. This is not true for the plans that we represent, since the effects of a plan do not include the effects of its sub-goals; hence a plan that consists only of sub-goals and no actions, will not have any effects attached to it.

While the concept of in-conditions is not apparent in classical planning, it is used in some of the work in the theory of multi-agent systems (Tidhar, Rao, Ljungberg, Kinny & Sonenberg 1992) as well as in recent work in hierarchical task network planning (Clement & Durfee 1999*a*, Clement & Durfee 1999*c*, Clement et al. 2001). The details are found in the discussion below.

In the area of Hierarchical Task Network planning, the work of Clement et al. (1999*c*, 2001, 1999*a*, 1999*b*) use the concept of in-conditions along with pre-conditions and effects (which they call post-conditions), in defining the structure for a hierarchical plan. They define a plan as a tuple;

$\langle$ *pre, in, post, type, subplans, order* $\rangle$

The *pre, in* and *post* are the same as our pre-conditions, in-conditions and effects respectively. The plan *type* is one of *primitive*, *and* or *or*. An *and* plan is where all the *subplans* must be executed in the order specified by the ordering constraints *order*. If no order is specified the subplans are assumed to be concurrent. An *or* plan is where only one of the subplans need to be executed hence no ordering is needed. A *primitive* plan is one that specifies an atomic action and contains no subplans.

In comparison, in our approach, all the actions and sub-goals within a plan must be executed (“and”) and the order is defined in the way the actions and/or sub-goals are combined using the sequential (;) and parallel ($\|$) operators. Plan choice occurs for each (sub) goal where the agent needs to execute just one plan from the plans that satisfy the goal (“or”).

Clement et al. (2001) represent resources as they too reason about resource conflicts in their research. They, like us, categorize resources into *depletable* and *non depletable* resources, which are resources that cannot be re-used and those that can, respectively. They represent a resource by the following tuple:

$\langle$ *local_min_range, local_max_range, persist_range* $\rangle$

where for the local task the resource usage lies between the *local_min_range* and *local_max_range*. The *persist_range* is specific for depletable resources and specify the minimum and maximum amount of those resources that lasts after the task terminates. Since non-depletable resources are made available immediately after use, their persist_range is always (0,0).

The representation we use in this thesis for a particular resource, is simpler but adequate for our algorithms that manage resource conflicts. We define a resource as a $\langle$ *resource_name, value* $\rangle$ pair, where, *resource_name* is the name of the resource (e.g. energy), and *value* is the upper bound of the amount of the resource that may be required. So while Clement et al.'s work (Clement et al. 2001) provide a near optimal resource management framework with complex algorithms for managing resource conflicts, in this thesis we provide a possibly less optimal resource management mechanism but with very efficient algorithms that ensure avoidance of detrimental resource conflicts.

In the theory of multi-agent systems, Tidhar et al. in their work on team formation (Tidhar et al. 1992, Kinny, Ljungberg, Rao, Sonenberg, Tidhar & Werner 1992, Ljungberg, Rao, Sonenberg, Tidhar & Kinny 1992), provide a team representation language (Tidhar et al. 1992), which defines the plan of an agent as a tuple:

$$(p, \psi_{purpose}, \psi_{precond}, ML, p_{body}, AL, DL)$$

where *p* is the plan-type, $\psi_{purpose}$ is the goal that the plan achieves, $\psi_{precond}$ is the set of of pre-conditions, and *ML (Maintenance List)* is the set of in-conditions.

The plan body p_{body} is specified by a *plan graph* which is a rooted, directed, acyclic graph where each edge is labelled with a plan statement. These plan statements can be atomic statements (which are termed BASIC SKILLS) or a sub-goal (termed achievement states). The representation for the plan body within this thesis could also be viewed as a similar graph that is represented in a textual form. However, unlike in this thesis, Tidhar et al. do not allow plan statements to be executed in parallel.

Tidhar et al. represent the effects of a plan by *AL (Add List)* and *DL (Delete List),* which are lists of belief formulae that are added and deleted respectively from the agent's beliefs upon completion of the plan.

Practical implementations:

Practical BDI systems such as JACK (Busetta et al. 1999), dMARS (d'Inverno et al. 2004), PRS (Myers 1997), AgentSpeak (Rao 1996), JAM (Huber 1999) and Jason (Bordini & Hübner 2004) represent the following attributes of plans:

pre-conditions - are represented in all the mentioned systems as pre-conditions in PRS and JAM, and as *context conditions* in JACK, dMARS, AgentSpeak and Jason.

in-conditions - are represented in dMARS, JAM, JACK and PRS.

In dMARS they are termed *maintenance conditions,* and in JAM they are termed *context conditions*. This is unlike systems such as JACK, where context conditions are used to refer to what we term pre-conditions.

JACK and PRS have a more flexible form of in-conditions, where it is possible to place in-conditions on an arbitrary block of code within a plan. Note that this block of code may contain the entire plan body if required. In JACK, in-conditions can be placed by using the *@maintain* command (*@maintain(conditions-to-be-maintained, subtask)*). In PRS, a *REQUIRE_UNTIL (G C)* sub-goal can be used, which maintains condition *G* until condition *C* is made true.

effects - none of these systems explicitly state the effects of a plan. The effects are implicit in the form of beliefs that are added and/or deleted from the agent's belief set from within the plan.

plan body - in general, the plan body within these systems consist of plan statements, which can be atomic actions, sub-goals, statements that update the beliefs of the agent, or arbitrary code specific to an application. The syntax for actions, sub-goals, and belief updates differ between the systems.

Resources - from the systems mentioned, only PRS provides an explicit representation of resources at a plan level. They provide the *USE-RESOURCE (A B)* construct, which indicates that the resources *A* and *B* must be allocated to the plan for its entire duration, and cannot be allocated to another task. Furthermore, if these resources cannot be allocated to the plan before it begins, the plan is deemed to be not applicable. Apart from this allocation process, no further reasoning is done about resources within the PRS framework.

In addition to the above, JAM also defines a method (rather than an attribute) called *effects* within a plan. This is an optional function that allows the programmer to specify actions (usually belief updates) to occur when the plan succeeds. Similarly, JACK and dMARS define a *pass* and *fail* method, which specifies statements to be performed when a plan succeeds or fails respectively. PRS defines a *CONCLUDE* goal, which is a special type of goal in PRS that can be used as a sub-goal at the end of the plan to perform any belief updates that are necessary.

With the exception of JAM, none of the other BDI systems mentioned above represent goals explicitly. A discussion of JAM's representation of goals was presented before, in section 2.1.

3.3 The Goal-Plan Tree

Within the general BDI framework, goals are achieved by executing plans and each goal has at least one plan, if not many, that can be used to satisfy the goal. Each plan can include sub-goals, but need not have any. This relationship between a top level goal, its plans and sub-goals defines a tree structure for each top-level goal, which is termed the *goal-plan tree* for that goal. Following is a BNF definition of the goal-plan tree (GoalPlanTree), which is an extension to the BNF definition of the goal and plan structures presented in section 3.1.

⟨GoalPlanTree⟩ → ⟨GoalNode⟩
⟨GoalNode⟩ → ⟨GoalTypeName⟩ **{**⟨PlanNodes⟩**}**
⟨PlanNodes⟩ → ⟨PlanNode⟩ (**,** ⟨PlanNode⟩)*
⟨PlanNode⟩ → ⟨PlanTypeName⟩ ⟨GoalNode⟩ | ⟨PlanTypeName⟩ ⟨Operator⟩
⟨Operator⟩ → ⟨SequenceOperator⟩ | ⟨ParallelOperator⟩
⟨SequenceOperator⟩ → **{;** ⟨SubGoalTree⟩ **,** ⟨SubGoalTree⟩ (**,** ⟨SubGoalTree⟩)* **}**
⟨ParallelOperator⟩ → **{**|| ⟨SubGoalTree⟩ **,** ⟨SubGoalTree⟩ (**,** ⟨SubGoalTree⟩)* **}**
⟨SubGoalTree⟩ → ⟨GoalNode⟩ | ⟨Operator⟩

In this goal-plan tree, goals are represented as goal-nodes (GoalNode) and plans as plan-nodes (PlanNode). Each node contains a reference to the goal-type or plan-type that it represents. Therefore, the attributes of the relevant goal or plan can be accessed at each node.

The root of a goal-plan tree is a top-level goal, and its children are its potential plans. For each plan-node, its children are the sub-goals of the plan. Recall that the sub-goals may be combined in sequence or in parallel with each other. The *sequence-operator* (;) and *parallel-operator* (||) are used to denote this in the tree. Figure 3.1 is an example goal-plan tree for a goal to go to the beach (*GoToBeach-Goal*).

The agent only needs to execute one of the plans in order to satisfy the goal. Therefore, in the tree structure the children of a goal-node (i.e. the plan-nodes) are viewed as "OR" nodes. On the other hand, for a particular plan to succeed, all the sub-goals of a plan (if any), irrespective of whether they are in sequence or in parallel, must be attempted and succeed. Hence, the children of a plan-node (i.e. the goal-nodes) are viewed as "AND" nodes, in the tree. The leaf nodes of the tree are plan-nodes with no children. In other words, they are plans that have no sub-goals.

Actions are not represented within the tree structure, because they are considered to be atomic and their properties such as the pre-conditions, in-conditions and effects are abstracted to the plan level.

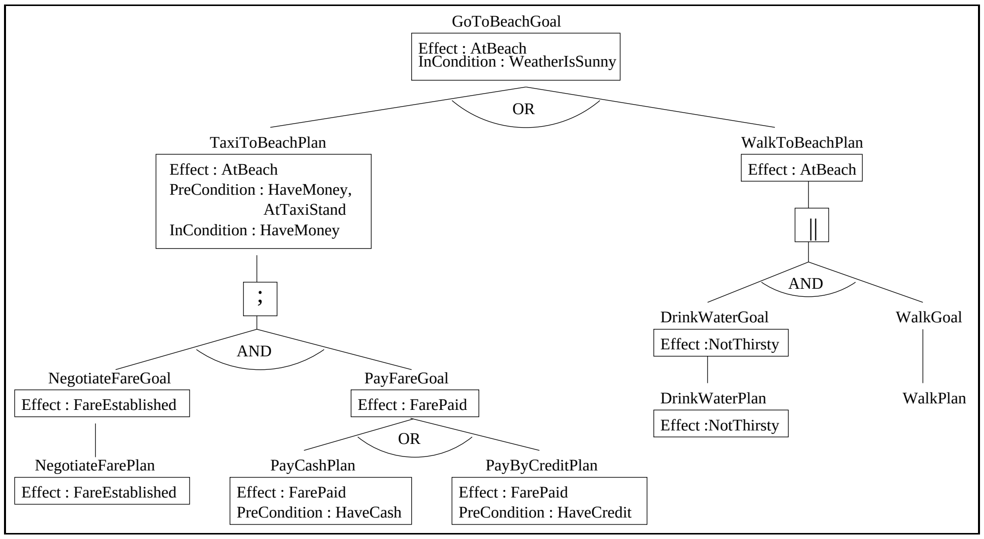

Figure 3.1: *An example of a goal-plan tree.*

When the agent system is compiled, the goal-plan tree templates for each goal are created and when the agent decides to pursue a goal, a goal-plan tree instance is created at run-time. As an agent executes its plans and sub-goals this tree is updated dynamically to reflect its progress. For example, if we assume the agent with the goal of *GoToBeachGoal* chose the *TaxiToBeachPlan* and has completed *NegotiateFarePlan* and begun the *PayByCreditPlan*, the tree would be annotated as shown in figure 3.2. This allows the agent to monitor the progress of its goals.

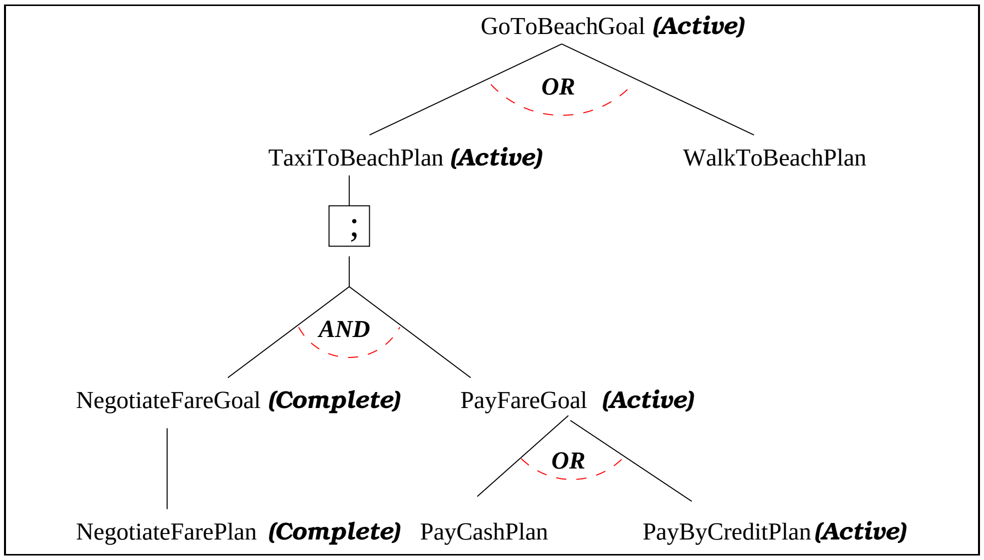
GoToBeachGoal ***(Active)***
OR
TaxiToBeachPlan ***(Active)***
WalkToBeachPlan
;
AND
NegotiateFareGoal ***(Complete)***
PayFareGoal ***(Active)***
OR
NegotiateFarePlan ***(Complete)***
PayCashPlan
PayByCreditPlan ***(Active)***

Figure 3.2: *Dynamically updating a goal-plan tree.*

One of the key features that this tree structure provides is that it allows information about (sub)goals and plans at the lower levels of the hierarchy to be propagated to higher levels all the way up to the root top-level goal-node. This propagated information stored at each node-level is termed *summary information*.

The summary information can be used to detect interactions between goals. For example, in the goal-plan tree for *GoToBeachGoal* (figure 3.1), it is possible to propagate the fact that *TaxiToBeachPlan* requires the pre-condition of *HaveMoney* to the top-level goal-node. The agent would then know that there is a *possibility* that it may require *HaveMoney* in order to satisfy *GoToBeachGoal*. It is only a *possibility* because it depends on the choice of plan. If *HaveMoney* was required no matter which way the agent chose to achieve *GoToBeachGoal*, then we say *HaveMoney* is *definitely* required.

If the agent then decides to pursue another goal that causes it to lose all its money (i.e. *not HaveMoney*), it can use the summary information to realise that there is a possible conflict. This does *not* require the agent to examine all the plans of each goal, the sub-goals of each plan and so on, which can be a very costly operation. This is the fundamental concept that is used in our algorithms that discover interactions between goals.

Apart from pre-condition, in-condition and effect summaries at each node, additional information is summarised according to the requirements of the reasoning methods and is described in detail, in the coming chapters.

3.4 Reasoning about Goals

An intelligent agent typically pursues multiple goals and when possible strives to achieve them simultaneously. When an agent has more than one goal to pursue, there are a number of ways in which the pursuit of these goals can fail to be independent. While the interaction between goals can lead to conflicts, they can also lead to situations where the agent could execute a single plan that satisfies a part of more than just one goal. This could be beneficial in terms of efficiency and resource usage.

Negative Interactions

Interactions that cause conflict are best detected prior to the agent pursuing its goals since these agents deliberate and act, and the actions are not always reversible, especially in the case where resources are consumed.

Resource conflict is common in applications where the agents are resource-bounded. For example, assume the agent has 100 units of energy and decides to pursue two goals, each requiring 70 units and 80 units respectively, at the same time. One possible sequence of events is shown in figure 3.3. In this situation neither of the goals succeed. However, if the agent had identified the conflict and chosen to pursue just one goal and not both then at least one goal may have completed successfully. While application specific mechanisms can be developed to manage resources, it is more useful to provide a generic mechanism which supports resource related reasoning at an infrastructure level, as we do in this thesis. In order to use this reasoning support that we provide, only the required declarative information needs to be specified at the application level without the need for application specific procedures that manage conflict.

When a goal is pursued by the agent there are sometimes conditions that must be protected during the execution of a goal/plan or else the goal/plan would fail. We have termed these conditions as in-conditions (see section 3.1). When a goal/plan is active, the agent must take care not to perform some other action that might threaten the in-conditions of the active goal/plan.

There is another type of condition that the agent must take care not to undo as we will now describe. In designing a recipe (i.e. a set of plans and sub-goals)

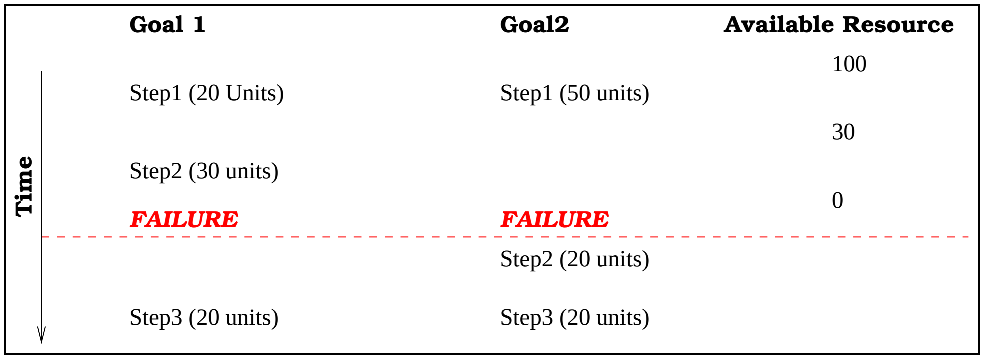

Figure 3.3: *An example of resource conflict.*

for achieving some goal, often early plans are introduced which set up (or achieve) effects that are required (as pre-conditions) by later plans. We shall term these effects as *preparatory effects*. In order for the overall recipe of achieving the goal to be successful, these preparatory effects when achieved should not be undone until the plans that require them begin execution. In order to facilitate this we provide mechanisms that identify preparatory effects and when active protect them by guarding against potentially harmful effects of other goals of the agent. Figure 3.4, shows an example where the effects of one goal threaten a preparatory-effect of another.

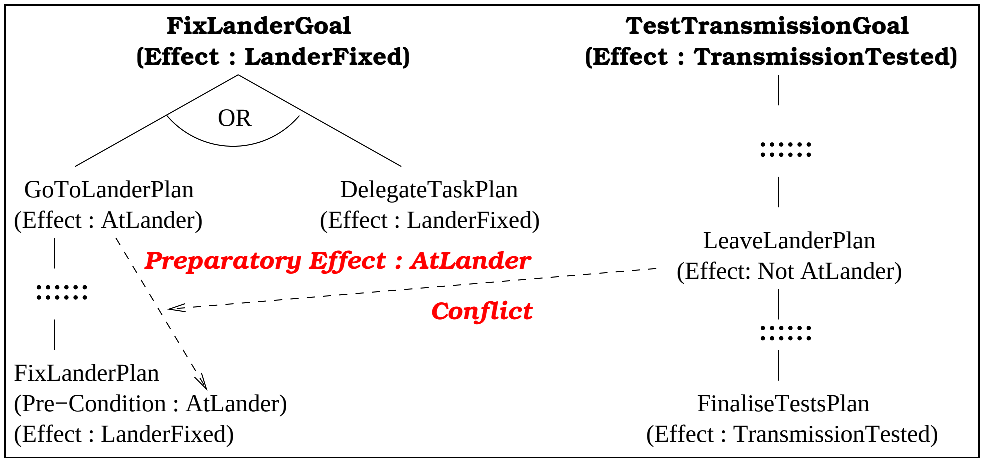

Figure 3.4: *An example of preparatory effects and related conflict.*

In the conflict identified in figure 3.4, there are two things that should be noted: the fact that the conflict would only occur if the agent chose *GoToLanderPlan* to satisfy *FixLanderGoal*, and *LeaveLanderPlan* to satisfy *TestTransmissionGoal*, and that the conflict occurs only during some part of the goal pursuit.

- The choice of plan is often determined by the context that the agent is in. However, it is possible for more than one plan to be applicable in a given context. In such a situation, it is clearly rational for the agent to choose the plan that potentially cause the least conflict (if any).

- A simple solution to satisfy goals that conflict because one goal's actions threaten a protected condition (i.e. in-condition or preparatory-effect) of another, is to execute the goals sequentially, completely independent of each other. However, in the case where the conflict is only between some particular parts of two top-level goals, it is more sensible for the agent to identify these parts and perform online scheduling to ensure they occur independent to each other. The non-conflicting parts can still be attempted in parallel.

When there is no conflict present, the agent should be able to detect this and pursue the goals in parallel. That is, there is no interference at all between the goals. Hence, the plans to achieve each goal can be freely interleaved.

Chapters 4 and 5, address these issues of negative interactions identified above.

Positive Interactions

Goals can also interact positively by assisting in the achievement of other goals. One common type of positive interaction is when two goals have common steps that can be merged. Figure 3.5, shows an example where plans of two goals could be merged. Another form of positive interaction as investigated by Horty and Pollack (2001) is when the agent's current goals increase the utility or preference of a new goal that it is being considered (see section 2.7). For example, the preference to buy a shirt would increase if the agent already has a goal to buy a tie, if we assume that they both require going to the same clothes shop. We do not reason about this type of interaction in this thesis.

Our focus is on identifying situations where plans of two goals could potentially be merged and providing mechanisms that facilitate this merge. The agent facilitates merging of plans by synchronising the two goals' progress such that the plans to be merged are executed at the same time. The reason that the two plans can be merged is because they both achieve the same effect. In the example shown in figure 3.5, the agent could choose to execute any one of the two plans as a result of the merge as they both achieve only one common effect. There can however be situations where the choice is not so trivial. For example, if *StrollToLanderPlan* achieved another effect *BatteryCharged* in addition to *AtLander* then although

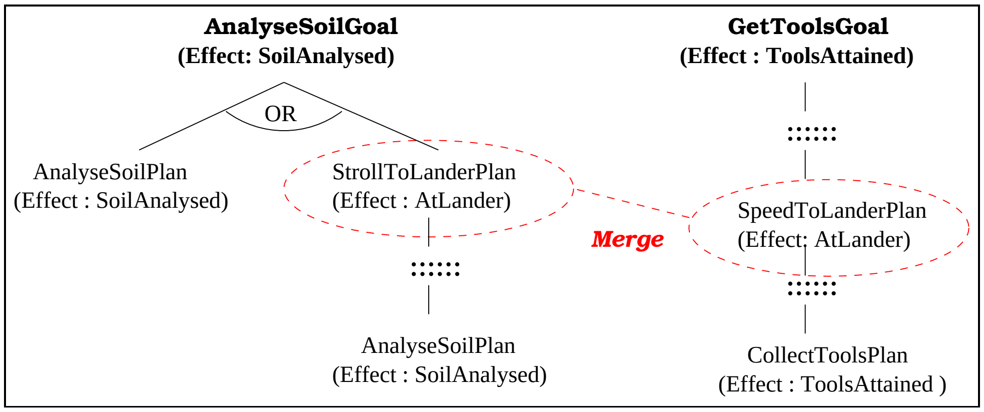

Figure 3.5: *An example of a potential plan merge.*

StrollToLanderPlan and *SpeedToLanderPlan* could still be merged, the plan that gets executed as a result of the merge will be *StrollToLanderPlan* since the other does not achieve *BatteryCharged*. Therefore, in the case of positive interactions, the agent must first detect the possibility of plan merges, schedule the plans in such a way that the plans to be merged are synchronised, and then reason about which plan to execute as a result of the merge.

We reason about positive interactions as described above, in chapter 6.

Chapter 4

Resource Conflicts [†]

Agents created in real world applications may use various kinds of resources to satisfy their goals. For example, a Mars rover agent with the goal of performing experiments on soil samples on the surface of Mars may require resources such as energy (fuel) to power on-board machinery, and a communication channel to transmit experimental results.

These agents pursue multiple goals, some being pursued concurrently. The resources available to these agents are usually finite. When multiple goals compete for these finite resources, conflicting situations are not uncommon. For example, two goals that require 100 units of energy each cannot be successfully pursued together, if only 150 units of energy are available. There is a risk that if both goals are attempted neither may succeed. A rational agent should identify such conflicts and at least not attempt to pursue goals that will inevitably conflict.

In the previous chapter, we presented structures for goals and plans of an agent. In this chapter we use these structures, and further develop algorithms and additional data structures that will allow the detection of resource conflicts between goals of an agent.

The first section of this chapter categorizes different resource types which influence the reasoning algorithms. The following sections present mechanisms that:

- allow agents to be aware of the resource needs of pursuing a goal;
- use these resource needs to identify potential resource conflicts; and
- allow intelligent agents to manage potential conflicts over resources, either by not pursuing some goals to allow others to be successful or, where possi-

[†] Part of the work presented in this chapter has been previously published (Thangarajah, Winikoff, Padgham & Fischer 2002).

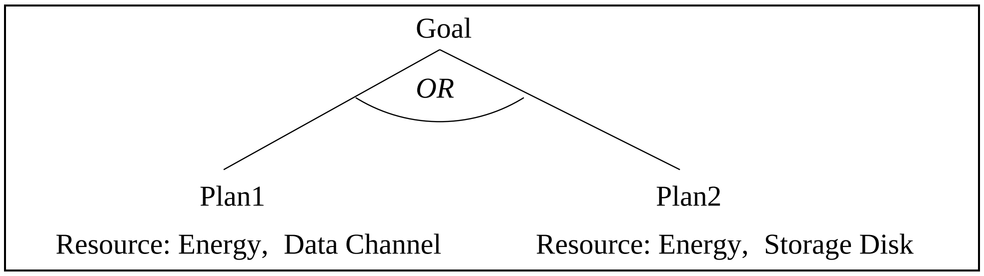
Goal
OR
Plan1
Plan2
Resource: Energy, Data Channel
Resource: Energy, Storage Disk

Figure 4.1: *An example of necessary and possible resource types.*

ble, by scheduling the plan execution to avoid conflicts. For example, consider an autonomous robot with 10 units of energy, with two goals to pursue, one of which requires 8 energy units, the other 5. It would not be rational for the robot to pursue both of these goals, as it is obvious that at least one would fail. If this same robot were to pursue two other goals where each required a communications channel, and only one communication channel was available, an intelligent robot may conclude that it could pursue both, by scheduling the use of the resource.

4.1 Resource Types

Necessary and Possible Resources

In BDI systems, there are often a number of different ways of achieving a particular goal. The resource requirements may depend on the chosen method. As different plans are typically appropriate for different environmental situations, the choice of plan is likely to depend on the context in which the agent is situated. Consequently, it is not always possible to state in advance precisely what resources will be required to accomplish a given goal. However, it may be possible to deduce which resources are definitely required and which are possibly required to achieve a goal.

If a particular resource will be needed in every way that the agent knows how to achieve a given goal, this resource is classified as a *necessary* resource. A resource that is not necessary, but may be required depending on the choice of plan, is classified as a *possible* resource. In the example illustrated in figure 4.1 above, *energy* is a necessary resource, *data channel* and *storage disk* are possible resources.

It should be noted that having all necessary resources does not mean that sufficient resources are available to achieve the goal. In the above example, although *energy* is the necessary resource, that alone is not sufficient to execute either of the plans. The sufficient resources will be *energy* and at least one of *data channel* or *storage card* (i.e. $energy \wedge (data\ channel \vee storage\ card)$). Representing sufficient resources will require the use of disjunctive conditions, which will inevitably lead to an explosion in computational complexity. Consequently, the reasoning in this thesis is limited to detection of straightforward conflicts with respect to each individual resource, and not to reason about satisfiability. Satisfiability can be assumed only if all possible resources are available. The algorithms that we develop in this chapter do not require knowledge of sufficient resource needs, and only require the necessary and possible resource categorizations which provide the lower and upper bounds of resource needs.

Consumable and Re-usable Resources

We also classify resources into *re-usable* and *consumable* resources depending upon their ability to be re-used. Resources that are no longer available after use are termed *consumable* resources. For example, *energy* is a consumable resource. If an agent has 100 units of energy available before a plan begins and the plan utilises 30 units of energy, then only 70 units of energy are available after the plan is completed. The agent may however, have some means of replenishing these resources, which would increase the amount of resources available.

Resources that can be made available after use are termed *re-usable* resources. For example, a communication channel is a re-usable resource. If an agent has one data channel before a plan begins and the plan requires a dedicated data channel, this data channel would be in use during the plan execution and unavailable to other plans. However, after the plan completes, the data channel is available for use by other plans if needed.

Example

The following example is used throughout the sections in this chapter to illustrate more clearly the representations and algorithms as they are presented.

Consider a rover agent which is deployed on the Martian surface. The agent obtains energy from solar panels and is given requests to perform experiments on various substances (e.g. *Soil*, *Rock*). Two resources arising naturally in this domain are:

- **Energy -** This is a consumable resource (which can be renewed by basking in the sun at suitable hours of the day).
- **Communication channel -** This is a re-usable resource (which is used for

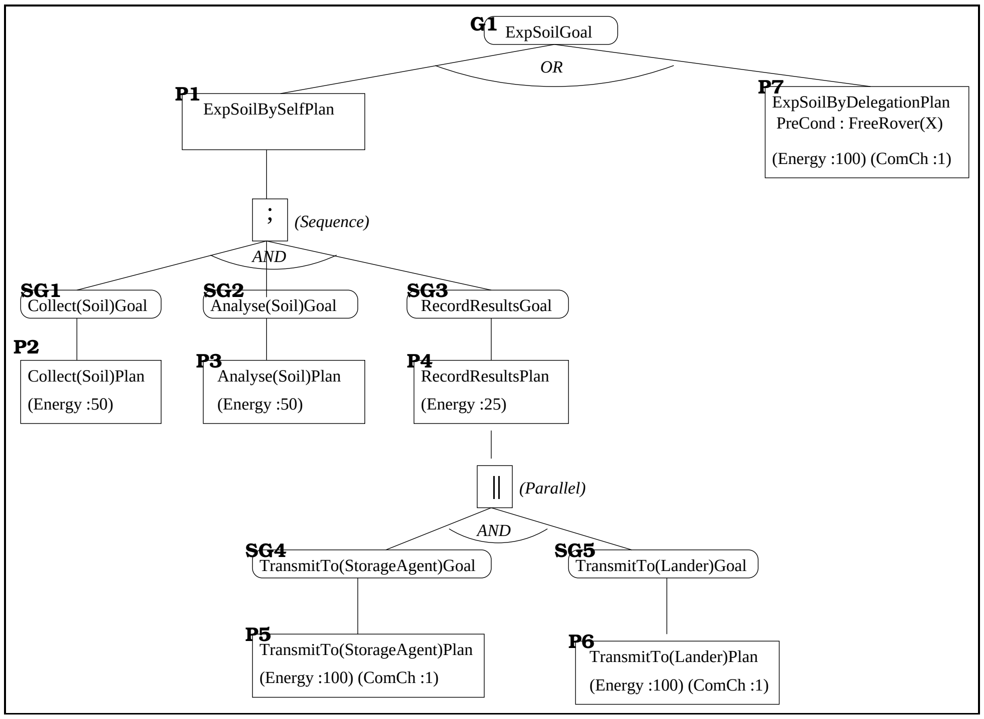

Figure 4.2: *A goal-plan tree for goal* ExpSoilGoal*.*

communicating with other rover agents and transmitting results back to the control station).

One of the goals that the rover agent may undertake is a goal to analyse soil samples (*ExpSoilGoal*). A possible goal-plan tree for this goal is shown in figure 4.2. In this example the agent has two plans that will achieve the goal: by doing it itself, or by delegating the task to another agent. A pre-condition to delegating the task is that there should be another rover agent that is not busy.

The resource requirements of each plan are annotated to the plan nodes in the goal-plan tree. Note that *ComCh* is an abbreviation for *Communication Channel* and resources are represented as *(name:value)* pairs, where *name* is the name of the resource (e.g. energy) and *value* is the maximum amount of that resource required by the plan (e.g. 100).

4.2 Representation of Resources

In order to reason about resource needs and how these affect the pursuit of goals, a representation of resources which supports the desired reasoning must be developed. In this section we provide a representation of resource related data structures and operators, which are used in the algorithms described in later sections to derive resource summaries as well as to detect resource conflicts.

Resource types:
A set of *resource types* is represented by $\mathcal{T} = \{t_1, \ldots, t_n\}$. For example, resource types required by the Mars rover agent ($\mathcal{T}_{\text{rover}}$) would be $\mathcal{T}_{\text{rover}} = \{\textit{Energy}, ComCh\}$.

Resource requirement:
A *resource requirement* is a pair of a resource type and a number, (t, n), where $n \geq 0$ and represents an amount of the resource. For example, $(\textit{Energy}, 200)$.

Resource set:
A *resource set* (R) is defined as a set of resource requirements. In the definitions ahead it is assumed that resource sets are *normalised* so that each resource type appears *exactly* once. For example, given the resource types above for the Mars rover agent, the resource set $\{(\textit{Energy}, 20), (\textit{Energy}, 30)\}$ is normalised to $\{(\textit{Energy}, 50), (ComCh, 0)\}$. In future, the presence of the resource types that have a value of 0 in the normalised resource set, is assumed and not explicitly included for clarity.

Re-usable and consumable resources:
Reusable resources of resource set R are denoted by R^r and the *consumable resources* by R^c. A resource set consists of both re-usable and consumable resources, $R = R^c \cup R^r$, and $R^c \cap R^r = \emptyset$
The re-usable resources and consumable resources are mutually exclusive (i.e a resource cannot be consumable as well as re-usable).

$\sqsubseteq$ operator:
In order to compare two sets of resources in terms of resource availability, the *smaller than or equal to* operator $\sqsubseteq$ is introduced. Resource set R_1 is smaller than or equal to resource set R_2 (written $R_1 \sqsubseteq R_2$) if any resource type that appears in R_1 also appears in R_2 and if whenever a resource type t appears in R_1, the amount required is less than or equal to the amount required in R_2. Formally (recall that normalised resource sets are assumed):
$R_1 \sqsubseteq R_2$ iff $\forall t. \#(R_1, t) \leq \#(R_2, t)$

where $\#(R, t)$ denotes the amount of resource type t in R.
For example, if R_1 = {(energy, 50), (ComCh, 1)} and R_2 = {(energy, 150), (ComCh, 1)} then $R_1 \sqsubseteq R_2$.
Note that $\sqsubseteq$ is a partial order (reflexive, transitive, antisymmetric). $R_1 \sqsubset R_2$ is used to denote $R_1 \sqsubseteq R_2 \wedge R_1 \neq R_2$.

Resource summary:
A resource summary, $\mathcal{S}$, consists of two resource sets of resource requirements,
$\mathcal{S} = \langle N, P \rangle$, where N represents the *necessary* resource set and P represents the *possible* resource set. Note that P is always greater than N, that is any resource which is necessary for successful execution is also possibly required (formally $N \sqsubseteq P$). Therefore, N gives a lower bound of the resources required, while P gives an upper bound. The following section provides details on how resource summaries may be derived for goals and plans using the goal-plan tree structure.

4.3 Deriving Resource Requirements

In this section we describe the process of determining the resource summary ($\mathcal{S}$) for each plan and goal within the agent system. This derived resource summary can be used to determine various levels of potential conflict, as well as situations when it is safe to pursue given goals in parallel.

Resource requirements for a goal are derived by combining the resource requirements of all relevant plans for that goal. The resource requirements of a plan are calculated by combining resource requirements of the sub-goals and actions within the plan body.

Recall that resource requirements are not attached to each individual action in the system, but a summary of the resource requirements of all the actions within a plan is specified by the programmer at each plan[1]. These shall be referred to as the resource requirement of the plan, and denoted by the symbol $P^{\mathcal{R}}$ for a plan P. Note that $P^{\mathcal{R}}$ does not contain the resource requirements of the sub-goals within the plan, but only of its actions. For example, the resource requirements of the Mars rover agent's plan to transmit the results to the lander (*TransmitTo(Lander)Plan*) are {(Energy,50)}
i.e. $TransmitTo(Lander)Plan^{\mathcal{R}} = \{(Energy, 50)\}$

In order to facilitate the computation of resource summaries of goals and plans, a number of operators over resource sets need to be defined.

[1]The reason for this is that it gives a better mapping to implementations, where a plan may consist of arbitrary code plus sub-goals, rather than the simpler formalisation given here. It also relieves the programmer from the need to specify resource requirements at the level of each action.

$\sqcup$ and $\sqcap$ operators:
The $\sqcup$ operator computes the upper bound (maximum) of two resource sets. For example, if
$R_1 = \{(Energy, 50), (ComCh, 1)\}$ and $R_2 = \{(Energy, 100)\}$ then
$R_1 \sqcup R_2 = \{(Energy, 100), (ComCh, 1)\}$.

The $\sqcap$ operator computes the lower bound (minimum). Using the same example,
$R_1 \sqcap R_2 = \{(Energy, 50)\}$[2]
Formally:

$$\begin{aligned} R_1 \sqcup R_2 &= \{(t, \max(\#(R_1, t), \#(R_2, t))) \mid t \in \mathcal{T}\} \\ R_1 \sqcap R_2 &= \{(t, \min(\#(R_1, t), \#(R_2, t))) \mid t \in \mathcal{T}\} \end{aligned}$$

$\oplus$ and $\otimes$ operators:
When determining the resource needs of a plan, there is a need to distinguish between the sub-goals within a plan that are achieved in sequence and those that are achieved in parallel. If two goals G_1 and G_2 are done in parallel (i.e. $G_1 \parallel G_2$), then the resources of each goal can be combined by a simple addition of the resource sets, irrespective of whether the resources are consumable or re-usable. This operation is denoted by the $\oplus$ operator. For example, if the **necessary** resource requirements of G_1 and G_2 are N_1 and N_2 respectively and
$N_1 = \{(Energy, 50), (ComCh, 1)\}$ and $N_2 = \{(Energy, 100), (ComCh, 1)\}$ then

$$N_1 \oplus N_2 = \{(Energy, 150), (ComCh, 2)\}$$

If G_1 and G_2 are done in sequence (i.e. $G_1; G_2$), then addition would provide the correct combined requirement for resources that are consumable. However, because re-usable resources are available after use, the combined requirement of two sequential sub-goals is calculated using the upper bound ($\sqcup$) operator. This combination operation is denoted by the $\otimes$ operator.

$$N_1 \otimes N_2 = \{(Energy, 150), (ComCh, 1)\}$$

[2]Note that the result is the normalised form of $\{(Energy, 50), (ComCh, 0)\}$.

Formally, on any resource sets R_1 and R_2:

$$\begin{aligned} R_1 \oplus R_2 &= \{(t, \#(R_1, t) + \#(R_2, t)) \mid t \in \mathcal{T}\} \\ R_1 \otimes R_2 &= (R_1^c \oplus R_2^c) \cup (R_1^r \sqcup R_2^r) \end{aligned}$$

The **possible** resources of sub-goals are also determined in a similar manner. In general the resource summaries of two sub-goals that are executed in parallel are combined as:

$$\mathcal{S}(G_1 \| G_2) = \mathcal{S}(G_1) \oplus \mathcal{S}(G_2) = \langle N_1, P_1 \rangle \oplus \langle N_2, P_2 \rangle = \langle N_1 \oplus N_2, P_1 \oplus P_2 \rangle$$

and if they are executed in sequence as follows:

$$\mathcal{S}(G_1; G_2) = \mathcal{S}(G_1) \otimes \mathcal{S}(G_2) = \langle N_1, P_1 \rangle \otimes \langle N_2, P_2 \rangle = \langle N_1 \otimes N_2, P_1 \otimes P_2 \rangle.$$

$\uplus$ **operator:**
In computing the *possible* (maximum) resource requirements of a goal it is important to note that it may be necessary for more than one plan to be executed in order to achieve that goal, due to plan failure. Note also that consumable resources that have been used in this process cannot be recovered. Consequently, the possible resource needs of the goal must combine the resource needs of *all* relevant plans, as if each were to be executed sequentially applying the $\otimes$ operator as shown above.

The necessary resources of a goal are the minimum amount for each resource across *all* relevant plans of the goal and can be derived using the lower bound ($\sqcap$) operator.

The $\uplus$ operator can be defined to combine the resource requirements of plans as follows:

$$\langle N_1, P_1 \rangle \uplus \langle N_2, P_2 \rangle = \langle N_1 \sqcap N_2, P_1 \otimes P_2 \rangle$$

For example, if two plans PL_1 and PL_2 have the resource summaries of $S(PL_1)$ and $S(PL_2)$ respectively, and
$S(PL_1) = \langle \{(\textit{Energy}, 30)\}, \{(Energy, 50), (ComCh, 1)\} \rangle$ and
$S(PL_2) = \langle \{(\textit{Energy}, 20)\}, \{(\textit{Energy}, 20), (ComCh, 1)\} \rangle$ then

$$S(PL_1) \uplus S(PL_2) = \langle \{(\textit{Energy}, 20)\}, \{(\textit{Energy}, 70), (ComCh, 1)\} \rangle.$$

In the above case, the necessary energy is 20 units because that is the minimum energy required to achieve the goal (i.e. by choosing the plan PL_2). The possible energy required is 70 units because if the plan PL_2 is chosen and fails, and is then followed by the execution of the plan PL_1, the total possible units of energy required is 70 units. Since the resource $ComCh$ is a re-usable resource, even though it is possibly required by both PL_1 and PL_2, it can only be required by one at a time, as the plans are alternatives.

Having defined all of the above operators, we now define how the resource summaries of a goal can be computed, by combining the resource summaries of its components. The lifting of the operators $\oplus, \otimes, \sqcap$, and $\sqcup$ is defined to operate on pairs of resources in the obvious way. For example,
$\langle N_1, P_1\rangle \oplus \langle N_2, P_2\rangle = \langle N_1 \oplus N_2, P_1 \oplus P_2\rangle$.
The resource requirements of a goal G and a plan PL are denoted by $\mathcal{S}(G)$ and $\mathcal{S}(PL)$ respectively. Note that they are each a two tuple $\langle N, P\rangle$. The set of relevant plans for a goal G is denoted by *plans-of*(G). $PL^{\mathcal{R}}$ contains the resource requirements of all the actions within the plan body of PL and hence the resource requirements of individual actions are not considered in the function below. Also, recall from our representation for plans (see section 3.1) that sub-goals within a plan body can be either in sequence (;), in parallel (||), or in a nested combination of the two.

$$\begin{aligned}
\mathcal{S}(G) &= \biguplus_{p \in plans-of(G)} \mathcal{S}(p) \\
\mathcal{S}(PL) &= \langle PL^{\mathcal{R}}, PL^{\mathcal{R}}\rangle \oplus \mathcal{S}(PL.planBody)) \\
\mathcal{S}(G_1 \| G_2) &= \mathcal{S}(G_1) \oplus \mathcal{S}(G_2) \\
\mathcal{S}(G_1; G_2) &= \mathcal{S}(G_1) \otimes \mathcal{S}(G_2))
\end{aligned}$$

Complete example:
We will now derive the resource summary for the Mars rover agent's goal of analysing soil samples (*ExpSoilGoal*) as shown in figure 4.2, to illustrate the manner in which the above function operates.

For clarity, rather than referring to the names of each plan and goal in the example, we will refer to the label that is indicated in the figure. For example, $P2$ is used to refer to *Collect(Soil)Plan*. As the function is recursive, we will begin at the leaf nodes where the plans do not contain any sub-goals. $P2$, $P3$, $P5$, $P6$ and $P7$ fall into this category. Since these plans have no sub-goals, for each plan the $\mathcal{S}(body(PL))$ = $\langle\emptyset, \emptyset\rangle$. Therefore, for such plans the resource summary is simply the resource requirements of the plan. For example, for the plan $P2$:
$\mathcal{S}(P2) = \langle P2^{\mathcal{R}}, P2^{\mathcal{R}}\rangle = \langle\{(Energy, 50)\}, \{(Energy, 50)\}\rangle$
Similarly,
$\mathcal{S}(P3) = \langle\{(Energy, 50)\}, \{(Energy, 50)\}\rangle$
$\mathcal{S}(P5) = \langle\{(Energy, 100), (ComCh, 1)\}, \{(Energy, 100), (ComCh, 1)\}\rangle$

$\mathcal{S}(P6) = \langle\{(Energy, 100), (ComCh, 1)\}, \{(Energy, 100), (ComCh, 1)\}\rangle$
$\mathcal{S}(P7) = \langle\{(Energy, 100), (ComCh, 1)\}, \{(Energy, 100), (ComCh, 1)\}\rangle$

Next we look at sub-goals $SG1$, $SG2$, $SG3$, $SG4$ and $SG5$. Each of these sub-goals has just one plan by which it can be achieved; therefore the resource summary of each sub-goal is the resource summary of that particular plan.
$\mathcal{S}(SG1) = \mathcal{S}(P2) = \langle\{(Energy, 50)\}, \{(Energy, 50)\}\rangle$
$\mathcal{S}(SG2) = \mathcal{S}(P3) = \langle\{(Energy, 50)\}, \{(Energy, 50)\}\rangle$
$\mathcal{S}(SG4) = \mathcal{S}(P5) = \langle\{(Energy, 100), (ComCh, 1)\}, \{(Energy, 100), (ComCh, 1)\}\rangle$
$\mathcal{S}(SG5) = \mathcal{S}(P6) = \langle\{(Energy, 100), (ComCh, 1)\}, \{(Energy, 100), (ComCh, 1)\}\rangle$
$\mathcal{S}(SG3) = \mathcal{S}(P4) =$ yet to be derived

The plan $P4$ has two *parallel* sub-goals, $SG4$ and $SG5$. The resource summary of $P4$ is the combination of the resource summaries of its sub-goals and the resource requirements of the plan.

$$\begin{aligned}
\mathcal{S}(P4) &= \langle P4^{\mathcal{R}}, P4^{\mathcal{R}}\rangle \oplus \mathcal{S}(SG4 \| SG5)\\
&= \langle P4^{\mathcal{R}}, P4^{\mathcal{R}}\rangle \oplus \mathcal{S}(SG4) \oplus \mathcal{S}(SG5)\\
&= \langle P4^{\mathcal{R}}, P4^{\mathcal{R}}\rangle \oplus \ \langle\{(Energy, 100), (ComCh, 1)\}, \{(Energy, 100), (ComCh, 1)\}\rangle\\
&\qquad\qquad \oplus \langle\{(Energy, 100), (ComCh, 1)\}, \{(Energy, 100), (ComCh, 1)\}\rangle\\
&= \langle P4^{\mathcal{R}}, P4^{\mathcal{R}}\rangle \oplus \langle\{(Energy, 200), (ComCh, 2)\}, \{(Energy, 200), (ComCh, 2)\}\rangle\\
&= \langle\{(Energy, 25)\}, \{(Energy, 25)\}\rangle \oplus\\
&\qquad\qquad \langle\{(Energy, 200), (ComCh, 2)\}, \{(Energy, 200), (ComCh, 2)\}\rangle\\
&= \langle\{(Energy, 225), (ComCh, 2)\}, \{(Energy, 225), (ComCh, 2)\}\rangle
\end{aligned}$$

The plan $P1$ contains three *sequential* sub-goals and the plan itself has no resource requirements. The resource summary of the plan $P1$ is derived as follows:

$$\begin{aligned}
\mathcal{S}(P1) &= \mathcal{S}(SG1; SG2; SG3)\\
&= \mathcal{S}(SG1) \otimes \mathcal{S}(SG2) \otimes \mathcal{S}(SG3)\\
&= \langle\{(Energy, 50)\}, \{(Energy, 50)\}\rangle \otimes \langle\{(Energy, 50)\}, \{(Energy, 50)\}\rangle\\
&\quad \otimes \langle\{(Energy, 225), (ComCh, 2)\}, \{(Energy, 225), (ComCh, 2)\}\rangle\\
&= \langle\{(Energy, 325), (ComCh, 2)\}, \{(Energy, 325), (ComCh, 2)\}\rangle
\end{aligned}$$

Finally the resource summary of the goal $G1$ is derived by combining the summary of the two plans that could satisfy the goal: $P1$ and $P7$.

$$\begin{aligned}
\mathcal{S}(G1) &= \mathcal{S}(P1) \ \uplus \ \mathcal{S}(P7)\\
&= \langle\{(Energy, 325), (ComCh, 2)\}, \{(Energy, 325), (ComCh, 2)\}\rangle \ \uplus\\
&\qquad \langle\{(Energy, 100), (ComCh, 1)\}, \{(Energy, 100), (ComCh, 1)\}\rangle\\
&= \langle\{(Energy, 100), (ComCh, 1)\}, \{(Energy, 425), (ComCh, 2)\}\rangle
\end{aligned}$$

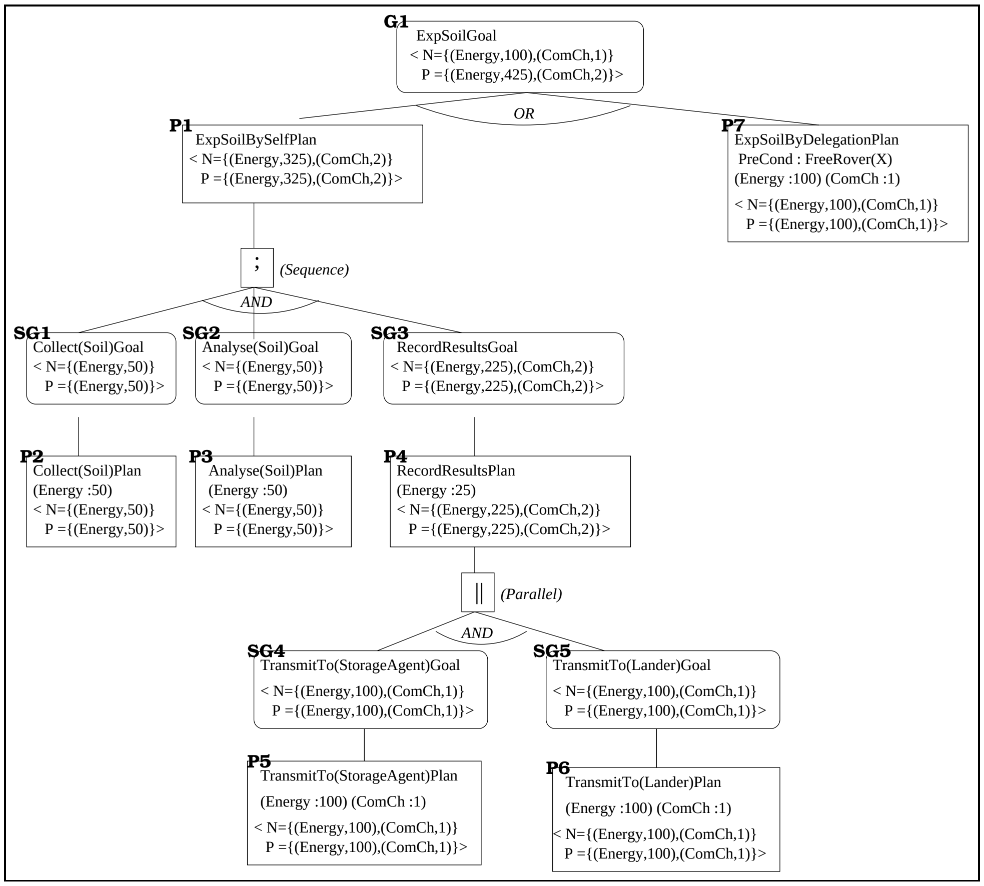
G1
ExpSoilGoal
< N={(Energy,100),(ComCh,1)}
P ={(Energy,425),(ComCh,2)}>
OR
P1
ExpSoilBySelfPlan
< N={(Energy,325),(ComCh,2)}
P ={(Energy,325),(ComCh,2)}>
P7
ExpSoilByDelegationPlan
PreCond : FreeRover(X)
(Energy :100) (ComCh :1)
< N={(Energy,100),(ComCh,1)}
P ={(Energy,100),(ComCh,1)}>
;
(Sequence)
AND
SG1
Collect(Soil)Goal
< N={(Energy,50)}
P ={(Energy,50)}>
SG2
Analyse(Soil)Goal
< N={(Energy,50)}
P ={(Energy,50)}>
SG3
RecordResultsGoal
< N={(Energy,225),(ComCh,2)}
P ={(Energy,225),(ComCh,2)}>
P2
Collect(Soil)Plan
(Energy :50)
< N={(Energy,50)}
P ={(Energy,50)}>
P3
Analyse(Soil)Plan
(Energy :50)
< N={(Energy,50)}
P ={(Energy,50)}>
P4
RecordResultsPlan
(Energy :25)
< N={(Energy,225),(ComCh,2)}
P ={(Energy,225),(ComCh,2)}>
||
(Parallel)
AND
SG4
TransmitTo(StorageAgent)Goal
< N={(Energy,100),(ComCh,1)}
P ={(Energy,100),(ComCh,1)}>
SG5
TransmitTo(Lander)Goal
< N={(Energy,100),(ComCh,1)}
P ={(Energy,100),(ComCh,1)}>
P5
TransmitTo(StorageAgent)Plan
(Energy :100) (ComCh :1)
< N={(Energy,100),(ComCh,1)}
P ={(Energy,100),(ComCh,1)}>
P6
TransmitTo(Lander)Plan
(Energy :100) (ComCh :1)
< N={(Energy,100),(ComCh,1)}
P ={(Energy,100),(ComCh,1)}>

Figure 4.3: *Goal-plan tree with annotated resource summaries for goal* ExpSoil-Goal.

These summaries are derived and attached to the goal-plan tree for each goal at compile-time. The complete annotated goal-plan tree for *ExpSoilGoil* is illustrated in figure 4.3.

Algorithms for Deriving Resource Summaries of a Goal-Plan Tree

The procedure *generateTree* creates a goal-plan tree for a particular goal and also derives and attaches the resource summary at each node.

```
procedure generateTree(Goal G, PlanLibrary P)
    GoalNode $G$ := new GoalNode(G)
    setTreeRoot($G$)
    addChildren($G$, P)

procedure setTreeRoot(GoalNode $G$)
    This procedure sets the goal $G$ as the root node of the goal-plan tree.

procedure addChildren(Node $G$, PlanLibrary P)
    selectedPlans($SP$) := selectPlans($G$, P)
    addToGoalNode($G$, $SP$)
    for each $p \in SP$
        subGoalTree($SGT$) := extractSubGoals($p$)
        if $SGT$ == empty then
            addToPlanNode($p$, $SGT$)
        else
            for each $sg \in SGT$
                addChildren($sg$, P)
            attachSummary( $p$, ($P^{\mathcal{R}}(p) \oplus \mathcal{S}'(SGT)$) )
    attachSummary($G$, $\mathcal{S}''(G)$)

function selectPlans(Goal $G$, PlanLibrary P) returns PlanSet $P$
    selects and returns the set of plans that satisfy the goal $G$

function extractSubGoals(Plan $p$) returns Node $n \in$ {goal-node, sequence-operator,
                                                          parallel-operator}
    Extracts the sub-goals from the plan body of $p$.
    If the plan has no sub-goals this procedure returns an empty tree (null in
    programming terms).
    If it has only one sub-goal, a node is created for that sub-goal (i.e. a goal-node)
    and returned.
    If the plan has more than one sub-goal, these sub-goals will be either in
    parallel, sequential, or in a nested combination of the two. Hence, this
    procedure would create and return a tree structure similar to figure 4.4,
    with the root node being either a sequence-operator or a parallel-operator.

procedure addToGoalNode(GoalNode $n$, PlanSet $C$)
    adds each plan of plan set $C$ as a child node to the goal node $n$ in the tree.

procedure addToPlanNode(PlanNode $n$, Node $sgt$)
    adds the node $sgt$ as a child node of $n$. Note that $sgt$ is
    either a single (sub) goal node or the head of a sub-goal tree.
```

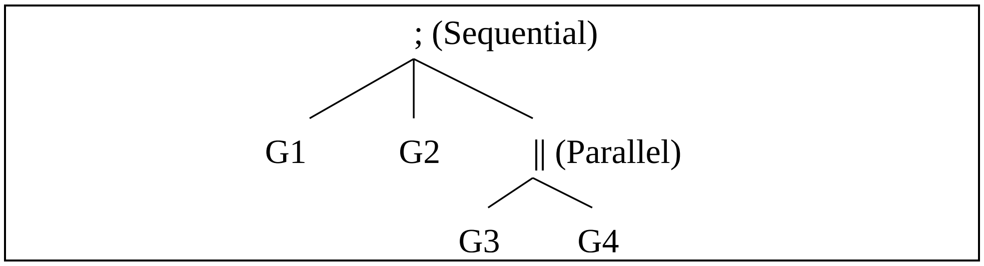

Figure 4.4: *A tree of sub-goals, which states that G1 is first executed followed by G2. After G2 completes, G3 and G4 are to be executed in parallel.*

```
procedure attachSummary(Node n, Summary S)
    attaches the summary S to node n in the tree

function $\mathcal{S}'$ (Node SGT) returns Summary S
    if SGT = GoalNode then
        return (SGT).summary
    if SGT = SequenceOperator (;) then
        return $\bigotimes_{node \in children\_of\_SGT} \mathcal{S}'(node)$
    if SGT = ParallelOperator (||) then
        return $\bigoplus_{node \in children\_of\_SGT} \mathcal{S}'(node)$

function $\mathcal{S}''$ (GoalNode G) : Summary S then
    return $\biguplus_{p \in goalnode.children} p$.summary
```

$\mathcal{S}'$ and $\mathcal{S}''$ only considers the resource summaries of the child nodes. This is efficient, since only the children of the node (that is the argument to these functions) are considered, only a single computation needs to be performed; namely combining the child nodes' summaries with the appropriate function ($\oplus$, $\otimes$, or $\uplus$, depending on the nature of node).

4.4 Resource Conflicts

In this section, mechanisms that detect resource conflicts will be presented using the derived resource summaries from the previous section.

We begin by providing some definitions relative to a set of goals. Given a set of goals $\{G_1, \ldots, G_n\}$ and the associated resource summary $\langle N_i, P_i \rangle$ for each goal G_i in the set, the following definitions are made:

1. the *sequential upper bound* $\bigotimes P_i$;[3]
2. the *parallel upper bound* $\bigoplus P_i$;[4]
3. the *sequential lower bound* $\bigotimes N_i$; and
4. the *parallel lower bound* $\bigoplus N_i$.

The $\otimes$ and $\oplus$ operators are as defined in section 4.3. Note that for sequential cases it is assumed that the re-usable resources may be re-used and for parallel cases no re-use is possible as the order of execution when in parallel cannot be guaranteed. These quantities are related as follows[5](see figure 4.5):

$$\bigotimes N_i \sqsubseteq \bigoplus N_i \sqsubseteq \bigoplus P_i \text{ and } \bigotimes N_i \sqsubseteq \bigotimes P_i \sqsubseteq \bigoplus P_i,$$

In general, $\bigotimes P_i$ and $\bigoplus N_i$ are not comparable.

It is possible to determine the nature of conflict within a goal set with respect to resources, by using these defined quantities and the available resources ($\mathcal{R}$). There are three primary cases: the available resources can be either less than the sequential lower bound, greater than or equal to the parallel upper bound, or in between the two.

- The first case, $\mathcal{R} \sqsubset \bigotimes N_i$, indicates that it is impossible to achieve all goals given the available resources, even if the goals were to be attempted in sequence. That is, the goals are definitely in *conflict*.
- The second case, $\bigoplus P_i \sqsubseteq \mathcal{R}$, indicates that there are sufficient resources to execute all goals even if the worst case scenario occurs for each goal. That is, the goals can be simultaneously executed in any way, conflict free; the goals are *safe*.
- The third case, ($\bigotimes N_i \sqsubseteq \mathcal{R} \sqsubset \bigoplus P_i$), indicates that there is uncertainty as to whether there are sufficient resources to pursue all goals. There are three sub-cases:
  - The first sub-case is where $\bigotimes P_i \sqsubseteq \mathcal{R} \sqsubset \bigoplus P_i$. Note that the difference between $\bigotimes P_i$ and $\bigoplus P_i$ is caused by the re-usable resources. So in this case there are enough resources to achieve the goals, but resource re-use is required. That is, the goals are *schedulable*: by

[3] $\bigotimes P_i = P_1 \otimes P_2 \otimes \ldots \otimes P_n$.

[4] $\bigoplus P_i = P_1 \oplus P_2 \oplus \ldots \oplus P_n$.

[5] Observe that $(A \otimes B) \sqsubseteq (A \oplus B)$ for any A and B, that $\otimes$ and $\oplus$ are both monotonic with respect to the $\sqsubseteq$ operator, and that $N \sqsubseteq P$.

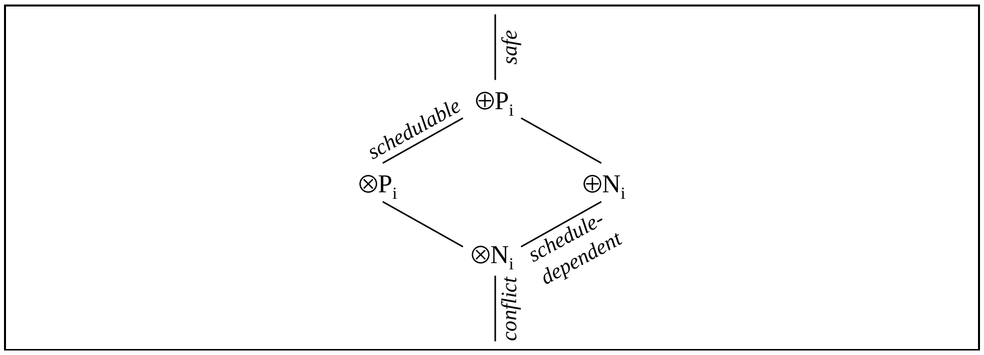

Figure 4.5: *The relationship between the status and the necessary and possible resource requirements of a goal set.*

scheduling[6] appropriately we can *guarantee* that there will be enough resources.

- The second interesting sub-case is where $\bigotimes N_i \sqsubseteq \mathcal{R} \sqsubset \bigoplus N_i$. In this case the necessary re-usable resources must be re-used, which requires the goals to be scheduled in order to ensure avoidance of any conflict. If the goals are not scheduled appropriately they will definitely fail. Therefore, this case is termed as *schedule-dependent*.
- The third sub-case occurs where the available resources do not fall into any of the above sub-cases (that is they are not safe, conflicting, schedulable or schedule-dependent). This status is called *uncertain*.

Note that the goal set may be both *schedulable and schedule-dependent* if $\bigotimes P_i \sqsubseteq \mathcal{R} \sqsubset \bigoplus N_i$[7]. These cases define how a set of goals can be compared to available resources to determine whether there are enough resources to pursue all goals and whether scheduling is necessary and/or possible.

A special situation of the above cases is where the "set" of goals consists of a single goal. In this situation while *safe* and *uncertain* maintain the same meaning:

- *conflicting* means that there are not enough resources available for the goal to succeed, and
- *schedulable* and *schedule-dependent* are not applicable as there is only one goal.

[6]Related work on scheduling in the area of databases and operating systems is discussed in the summary at the end of this chapter.

[7]The status "schedulable&schedule-dependent" is used to denote the case where both sub-cases apply.

The *status* (according to the cases above) of a set of goals $\mathcal{G}$ with respect to the available resources $\mathcal{R}$, is denoted as *status*($\mathcal{G}$, $\mathcal{R}$). The algorithm for this function is as follows:

```
function status ($\mathcal{G}$, $\mathcal{R}$)
    if $\bigoplus P_i \sqsubseteq \mathcal{R}$ then
        return safe
    else if $\mathcal{R} \sqsubset \bigotimes N_i$ then
        return conflict
    else if $\bigotimes P_i \sqsubseteq \mathcal{R} \sqsubset \bigoplus N_i$ then
        return schedulable&schedule-dependent
    else if $\bigotimes P_i \sqsubseteq \mathcal{R} \sqsubset \bigoplus P_i$ then
        return schedulable
    else if $\bigotimes N_i \sqsubseteq \mathcal{R} \sqsubset \bigoplus N_i$ then
        return schedule-dependent
    return uncertain
end function
```

Example:
The following example will illustrate the above algorithm. Assume that the agent has two goals G_1 and G_2 with the following resource summaries:

$\mathcal{S}(G_1) = \langle\{(\textit{Energy}, 110), (ComCh, 1)\}, \{(\textit{Energy}, 310), (ComCh, 2)\}\rangle$ and
$\mathcal{S}(G_2) = \langle\{(\textit{Energy}, 50), (ComCh, 1)\}, \{(\textit{Energy}, 90), (ComCh, 1)\}\rangle$.

We can then derive the following.

- $\bigoplus N_i = \{(\textit{Energy}, 160), (ComCh, 2)\}$
- $\bigotimes N_i = \{(\textit{Energy}, 160), (ComCh, 1)\}$
- $\bigoplus P_i = \{(\textit{Energy}, 400), (ComCh, 3)\}$
- $\bigotimes P_i = \{(\textit{Energy}, 400), (ComCh, 2)\}$

The *status* of the goal set $(\{G_1, G_2\}, \mathcal{R})$ under different resource availabilities $\mathcal{R}$, are shown below:

$\mathcal{R}$	status(G,$\mathcal{R}$)	reason
$\{(Energy, 600), (ComCh, 2)\}$	*Safe*[8]	$\bigoplus P_i \sqsubseteq \mathcal{R}$
$\{(Energy, 400), (ComCh, 2)\}$	*Schedulable*	$\bigotimes P_i \sqsubseteq \mathcal{R} \sqsubset \bigoplus P_i$
$\{(Energy, 160), (ComCh, 1)\}$	*Schedule-dependent*	$\bigotimes N_i \sqsubseteq \mathcal{R} \sqsubset \bigoplus N_i$
$\{(Energy, 100), (ComCh, 2)\}$	*Conflicting*	$\mathcal{R} \sqsubset \bigotimes N_i$
$\{(Energy, 600), (ComCh, 2)\}$	*Uncertain*	Not any of the above

Adding new goals into the system

Usually, instead of assessing the safety of a set of goals, it is often more desirable to determine the risks associated with adding a new goal or a set of goals G to an existing set of goals E. However, it is not possible to simply apply the definitions above without further requirements. The problem is that *if status*$(E, \mathcal{R})$ *is not safe to begin with, then status*$(E \cup G, \mathcal{R})$ *will not be safe, even if* G *uses no resources*. Therefore, it is not possible to determine the true status of adding G to E by simply applying the definitions we presented above.

The solution is to determine the risk of adopting G by only considering the resource types which are used by G. We define a notion of restriction – given resource sets R_1 and R_2, the restriction of R_1 by R_2 (written $R_1 \downarrow R_2$) is the resources in R_1 that are also used in R_2:

$$R_1 \downarrow R_2 = \{(t, \text{if } \#(R_2, t) > 0 \text{ then } \#(R_1, t) \text{ else } 0) \mid t \in \mathcal{T}\}$$

To determine the status of adding a new goal set to the existing set of goals, the definitions given above are used, but the resource sets are restricted before comparison. So, for example, if existing goals $E = \{G_1, G_2\}$ and the new goal being considered is G_{new} where $\mathcal{S}(G_{new}) = \langle N_{G_{new}}, P_{G_{new}} \rangle$, then the test for safety is whether

$$(P_{G_{new}} \oplus \bigoplus P_E) \downarrow P_{G_{new}} \sqsubseteq \mathcal{R}$$

and the check for conflict is whether

$$\mathcal{R} \sqsubset (N_{G_{new}} \otimes \bigotimes N_E) \downarrow N_{G_{new}}$$

The algorithm below computes the status of adding a new set of goals $\mathcal{G}$ to an existing set of goals, E, where resources $\mathcal{R}$ are available. $\mathcal{T}$ consists of all the resource types that are used by the existing goal set E. The algorithm considers each resource type of the resource set $\mathcal{T}$ separately. In doing so, for any resource *t* ($\in \mathcal{T}$), if the new goal set does not use that resource type then the goals are said to be *safe* with respect to that resource type.

If the necessary resource requirements of the new goal do not include the resource type *t*, but its possible resource requirements do, then the status obtained from the process defined above is revised to a more relaxed status as follows:

- If the status is derived to be *conflict* it is revised to *uncertain* because the

[8]This means that the set of goals is safe only with respect to resources. Failure can still be caused by other factors, including logical conflict.

necessary resource requirements do not interact. Definite conflicts can only be determined by examining the necessary resource requirements.

- Also, because the necessary resources do not interact, if the derived status is *schedulable&schedule-dependent* it is revised to *schedulable* (refer to figure 4.5), and
- if the status derived is *schedule-dependent* it is revised to *uncertain* (refer to figure 4.5).
- Since the *safe* and *schedulable* statuses do not require the comparison of necessary resources, the lack of such resources do not make any difference to these types of status.

After all resource types are computed the results are joined, yielding a status for the goal set $\mathcal{G}$. We define *status*$_t(\mathcal{G}, \mathcal{R})$ to be the same as *status*$(\mathcal{G}, \mathcal{R})$ except that all resource types other than t are ignored. The data structure *statusG* in the function is an array that stores the status of the goal set with respect to each resource type. That is, *statusG[t]* represents the status of adding $\mathcal{G}$ to E with respect to the resource type *t*.

If all the resource types have a *safe* status, the status of adding $\mathcal{G}$ to E is *safe*. On the other hand, if any of the resource types causes conflict, the overall status is set to *conflict*. If the statuses of the resource types do not fall into the above two categories, and there is at least one resource type with status *schedule-dependent*, the overall status is set to the same. This is because if it is not scheduled the goals may conflict with respect to that resource. If none of the above are true then the statuses of each resource type is either *safe, schedulable* or *schedulable&schedulable-dependent*. From them, if any are *schedulable&schedulable-dependent* then the status is set to the same, because this status also requires goals to be scheduled in order to avoid definite conflict. Otherwise, that is the statuses are either *schedulable* or *safe* for each resource type, the overall status is set to *schedulable*.

```
**function** *status*($\mathcal{G}$,$E$,$\mathcal{R}$)

    $\langle N_G, P_G \rangle$ := $\mathcal{S}(G)$
    **for each** resource type $t \in \mathcal{T}$
        **if** $\#(P_G, t) = 0$ **then** // the new goal does not use this resource
            statusG[t] := safe
        **else if** $\#(N_G, t) = 0$ **then** // no interaction with necessary resources
            **if** *status*$_t(E \cup \mathcal{G}, \mathcal{R})$=conflict **then**
            statusG[t]:=uncertain // necessary resources do not interact
        **else if** *status*$_t(E \cup \mathcal{G}, \mathcal{R})$=schedulable&schedule-dependent **then**
            statusG[t]:=schedulable
        **else if** *status*$_t(E \cup \mathcal{G}, \mathcal{R})$=schedule-dependent **then**
            statusG[t]:=uncertain
        **else**
            statusG[t] := *status*$_t(E \cup \mathcal{G}, \mathcal{R})$
    **else**
        statusG[t] := *status*$_t(E \cup \mathcal{G}, \mathcal{R})$
**endfor**

**if** $\exists t$ : statusG[t]=conflict **then** // at least one resource type causes definite conflict
    **return** conflict
**else if** $\forall t$ : statusG[t]=safe **then** // safe with respect to all resource types
    **return** safe
**else if** $\exists t$ : statusG[t] = schedule-dependent **then**
    // if at least one resource type must be scheduled to avoid conflict
    **return** schedule-dependent
**else if** $\forall t$ : statusG[t] $\in$ {safe,schedulable&schedule-dependent, schedulable} **then**
    **if** $\exists t$ : statusG[t]=schedulable&schedule-dependent **then**
        // the least safest of the statuses
        **return** schedulable&schedule-dependent
    **else** // the status of the resource types are either safe or schedulable
        **return** schedulable
**else**
    **return** uncertain

**end function**
```

This computation can be made more efficient by indexing on resource type. That is, for each resource type, a list of the (existing) goals which use that resource type can be maintained. Clearly, the algorithm can also be modified to return more detailed information such as, for each resource type, which goals in $\mathcal{G}$ use that resource, and which goals in E use the resource. This can then assist in dealing with conflicts.

The algorithms that we have presented, allow an agent to determine how safe it is to adopt a goal with respect to its existing set of active goals. This information allows the agent to be rational in its adoption of a goal by not adopting conflicting goal sets[9].

[9]It may also wish to reason about whether to drop existing goals but we do not deal with that here.

4.4.1 Dealing with Conflicts

If an agent determines that a new goal set $\mathcal{G}$ is *conflicting* with regards to its currently active goals E (and adoption will therefore inevitably cause the failure of some goal), then it is clearly not rational to simply adopt the new goal set. The rational behaviour is to not adopt $\mathcal{G}$, or to drop some existing goal(s) to give a revised set of existing goals $E^{'}$ such that $\bigotimes N_i \sqsubseteq \mathcal{R}$ for the set $\mathcal{G} \cup E^{'}$.

If the status of adding a new goal set is *schedulable* or *schedule-dependent*, the agent could simply attempt the goals sequentially. Whilst this is a simple solution it may be an over cautious approach since the resources that require scheduling may be needed by only some plans of the goals. In such situations, scheduling the plans that require these resources to occur in sequence is adequate, and other plans of the goals may be executed in parallel.

If the goals are attempted in parallel and the status is *schedule-dependent* then scheduling is *mandatory* in order to avoid conflict. For *schedulable* goals, scheduling can guarantee that enough resources will be available for the goals to be pursued without resource conflicts. However, in both cases, the agent also has the option of being *optimistic* by allowing the goals to be pursued without scheduling in the hope that an admissible order will naturally be achieved. The trade off between scheduling and not scheduling, is the cost of scheduling versus the risk of conflict. The risks associated with *schedule-dependent* goals are greater than *schedulable* since the former is guaranteed to fail if an admissible order is not achieved but the latter may still succeed.

In the case that the status of pursuing G with E is *uncertain* the agent has at least two choices. The first is to be *cautious* and always defer or reject the new goal set. The second is to be *optimistic* and always pursue the goals in parallel hoping that no conflict will occur. Both approaches are extreme and an alternative option would be for the agent to allow the goals to begin together, but *monitor* the progress of the goals as each step of a goal completes, looking for situations when the partially completed goals can be determined to be either definitely *safe* or definitely *conflicting*, in which case monitoring becomes no longer necessary. It is also useful to monitor situations where goals that were *uncertain* become *schedulable* or *schedule-dependent*, if scheduling is an option in the system.

4.5 Dynamic Updates of Resource Requirements

Given that the possible resource requirements of a goal considers the worst case situation, initially[10] there is likely to be a wide spread of values between the necessary

[10]that is, at compile time.

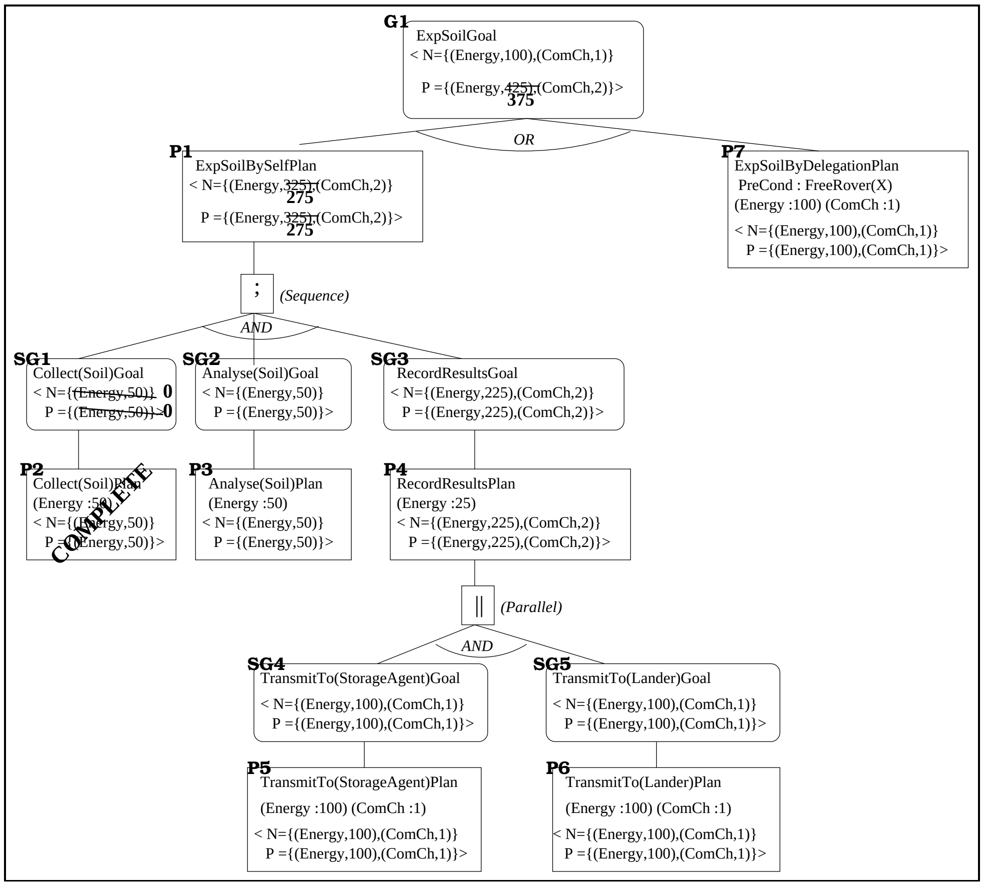
G1
ExpSoilGoal
< N={(Energy,100),(ComCh,1)}
P ={(Energy,~~425),~~(ComCh,2)}>
375
OR
P1
ExpSoilBySelfPlan
< N={(Energy,~~325),~~(ComCh,2)}
275
P ={(Energy,~~325),~~(ComCh,2)}>
275
P7
ExpSoilByDelegationPlan
PreCond : FreeRover(X)
(Energy :100) (ComCh :1)
< N={(Energy,100),(ComCh,1)}
P ={(Energy,100),(ComCh,1)}>
;
(Sequence)
AND
SG1
Collect(Soil)Goal
< N={~~(Energy,50)}~~ 0
P ={~~(Energy,50)}>~~ 0
SG2
Analyse(Soil)Goal
< N={(Energy,50)}
P ={(Energy,50)}>
SG3
RecordResultsGoal
< N={(Energy,225),(ComCh,2)}
P ={(Energy,225),(ComCh,2)}>
P2
Collect(Soil)Plan
(Energy :50)
< N={(Energy,50)}
P ={(Energy,50)}>
COMPLETE
P3
Analyse(Soil)Plan
(Energy :50)
< N={(Energy,50)}
P ={(Energy,50)}>
P4
RecordResultsPlan
(Energy :25)
< N={(Energy,225),(ComCh,2)}
P ={(Energy,225),(ComCh,2)}>
||
(Parallel)
AND
SG4
TransmitTo(StorageAgent)Goal
< N={(Energy,100),(ComCh,1)}
P ={(Energy,100),(ComCh,1)}>
SG5
TransmitTo(Lander)Goal
< N={(Energy,100),(ComCh,1)}
P ={(Energy,100),(ComCh,1)}>
P5
TransmitTo(StorageAgent)Plan
(Energy :100) (ComCh :1)
< N={(Energy,100),(ComCh,1)}
P ={(Energy,100),(ComCh,1)}>
P6
TransmitTo(Lander)Plan
(Energy :100) (ComCh :1)
< N={(Energy,100),(ComCh,1)}
P ={(Energy,100),(ComCh,1)}>

Figure 4.6: *Updated goal-plan tree for goal* ExpSoilGoal.

and possible resource requirements. This spread however, narrows dynamically as plans get executed and either succeed or fail. In this section we will describe a mechanism that allows the resource summaries to be dynamically updated at run-time.

In pursuing a goal G, once a plan P of G has completed, all possible and necessary resources associated with that plan are no longer relevant. Also with respect to the resource tree of G, when P completes, the summary information of the nodes (i.e. goal and plan nodes) above P in the tree will no longer be accurate. For example, in figure 4.3, if plan *Collect(Soil)Plan* completes, the resource summaries of the nodes above it will change as shown in figure 4.6 below.

Reasoning about the remaining resource requirements of partially achieved goals can enable an agent to make more up-to-date choices regarding its adoption of goals. For example, take the goal G_1 from figure 4.6, and a new goal the agent is considering, G_2, with resource summary $\langle\{\emptyset\}, \{(Energy, 75)\}\rangle$. If the current resource availability (i.e. after *Collect(Soil)Plan* of G_1 completes) is $\mathcal{R} = \{(Energy, 470)\}$ then using the updated resource summary of G_1 the agent can deduce that it is *safe* to execute G_2 in parallel with G_1 ($\oplus P_i = \{(Energy, 450)\} \sqsubset \mathcal{R}$). However, if the resource summary was not updated, then by using the original resource summaries the agent cannot deduce that it is safe to pursue the new goal ($\oplus P_i = \{(Energy, 500)\} \not\sqsubset \mathcal{R}$) which is not desirable.

This section describes a mechanism for maintaining information that efficiently calculates the remaining resource requirements for a partially completed goal.

Recall that the resource summary of each goal and its relevant plans are attached to the nodes of the goal-plan tree of the goal and that a template for this tree along with the resource summaries are created at compile time. At run-time each goal instance is assigned an instance of the associated goal-plan tree template. This tree can be updated at run-time by deleting nodes representing plans and subgoals as they complete, and recalculating resource summaries each time a node is deleted.

Although only the path from the root to the modified portion of the tree needs to be traversed for each update, this may be costly if done routinely (depending on the number of goals being pursued in parallel and the branching factor of the tree). This can be easily optimised by flagging the nodes as requiring update, but delaying the actual propagation of changes to resource summaries until there is some known need for monitoring or recalculation. This recalculation occurs when the agent considers adopting a new potentially non-safe goal.

An additional simple optimisation which we use, is to maintain a count at each node of the number of children that use each resource. If the node is a plan node whose $\mathcal{P}^R$ requires a resource *t*, then the count for *t* is 1 plus the number of children that require *t*. As a child node completes, this count is decremented. For example, in figure 4.6, at node G_1 the count for the resource *Energy* would be *6* initially (as there are 6 plans under the node that use it), but when plan $P2$ completes, the count will be decremented to *5*. When the count for a particular resource reaches zero, the need for that resource can be set to zero. In some cases this can allow faster updating and propagation, as once a resource need is zero or $P^{\mathcal{R}}$, remaining child nodes can be disregarded.

With these optimisations any cost beyond a couple of trivial operations is incurred only when a need is identified. The algorithm *delete* presented below, incorporates the above optimisations and provide a mechanism for updating the resource tree dynamically as plans and (sub) goals complete.

Updating the resource summary of a node (by the procedure *update*), sets the new value of the summary to the node and marks the parent of the node as *dirty*, which means that the parent's resource summary needs to be recomputed before use. This change is propagated all the way up to the root (using the procedure *propogateDirty*). The *update* procedure also decrements the usage count of the parent node (by the procedure *decrementCount*) and if the resource is no longer in use (i.e. usage count = 0) then the summary for that resource is set to zero. It is not indicated as dirty, as it is not necessary to recompute that resource type, since it is no longer in use.

When resource summaries are relied upon (for instance to detect conflicts), if the resources of concern are dirty then they need to be recomputed before use. The *recompute* procedure does this. We assume that *Summary*(t) returns a pair of records (necessary and possible) for the resource type t. For example, for goal G_1 in figure 4.6, *summary*$($*Energy*$) = \langle 100, 375\rangle_6$. Note that the subscript *6* in the notation denotes the usage count for that resource at goal G_1; there are 6 plans under that goal which require *Energy*.

Finally, recall that we assume that resource summaries are *normalised*.

Algorithm for updating the goal-plan tree with resource summaries

```
procedure delete(node)
    update(node, $\langle\emptyset,\emptyset\rangle$)
    if node $\neq$ root then // remove the node from the parent's child list
        node.parent.children := node.parent.children $\setminus$ node

procedure update(node,newSummary)
    if newSummary = node.summary then // no changes needed
        return
    for each $t \in$ resourcetypes
        $(n_{old}, p_{old})_{count}$ := node.summary($t$)
        $(n_{new}, p_{new})$ := newSummary($t$)
        if ($n_{old} \neq n_{new} \vee p_{old} \neq p_{new}$) $\wedge$ node $\neq$ root then
            if $n_{new} = p_{new} = 0$ then // resource no longer needed
                decrementCount(node.parent,$t$)
            else if node $\neq$ root
                node.parent.dirty[$t$] := true
                propagateDirty(node.parent, $t$)
    node.summary := newSummary
```

```
procedure propagateDirty(node, t)
    if node $\neq$ root $\wedge$ !node.parent.dirty[t] then
        node.parent.dirty[t] := true
        propagateDirty(node.parent, t)

procedure recompute(node)
    if $\neg\exists t$ such that node.dirty[$t$] = true then
        return // as there are no nodes that are dirty

    for each $c \in$ node.children
        recompute($c$) // if there are child nodes recompute them first

    if node is a GoalNode then
        node.summary := $\biguplus_{c \in node.children} c$.summary
    if node is a PlanNode then
        node.summary := node.plan.$P^{\mathcal{R}} \oplus$ node.child.summary
    if node is a ParallelOperator node (||) then
        node.summary := $\bigoplus_{c \in node.children} c$.summary
    if node is a SequenceOperator node (;) then
        node.summary := $\bigotimes_{c \in node.children} c$.summary

    for each $t$ such that node.dirty[$t$] = true
        node.dirty[$t$] := false // updated hence not dirty anymore

procedure decrementCount(node,t)
    $(n, p)_c :=$ node.summary(t)
    summary(t) := $(n, p)_{c-1}$
    if $(c - 1) = 0$ then // resource no longer in use
        summary(t) := $(0, 0)_0$
        if node $\neq$ root then
            decrementCount(node.parent, t)
    else
        node.dirty[t] := true
        if node $\neq$ root then
            propogateDirty(node.parent, t)
```

Summary

A rational agent should not adopt goals that clearly conflict. In this chapter we presented mechanisms that allow an agent to be aware of the possible and necessary resource requirements of goals, and to use these to detect resource conflicts. We provided algorithms that are computationally feasible, and that allow the agent to maintain up-to-date information with respect to the resource requirements of partially achieved goals.

The strengths of this approach of maintaining dynamic information about partially achieved goals are that, it allows us to avoid explicitly representing time, and it has efficient runtime algorithms and simple representations that are amenable to use in real applications. This is evident from the experimental work reported upon in chapter 7.

The area of conflict in multi agent systems has seen significant work (see for example (Tessier et al. 2000)). However, the focus has been on understanding the many types of conflict that can occur and how to resolve these conflicts rather than on providing algorithms and data structures that allow conflict detection. Also, much of the work has focussed on conflicts *between* agents, rather than on conflicts *within* an agent as explored in this chapter. The latter allows for considerably richer solutions since more information is available to the agent, and factors such as communication, shared knowledge and trust are not an issue.

In the area of databases and operating systems, resource conflict is also an issue (Silberschatz, Galvin & Gagne 2001). The “classical” approach to resource management in such areas is to use locking mechanisms, where a resource is allocated and locked before the steps that require it can occur. No other step can access this resource whilst it is locked. Situated intelligent agents cannot always use locking, and cannot always prevent conflict as the choice of plans cannot be determined in advance. However, since intelligent agents are designed to detect and recover from various failures, it is not necessary to guarantee that no conflict will occur. It is more important to detect conflicts so that appropriate recovery steps can be taken. Note that deadlock is not an issue since pre-emption is always possible (i.e. a plan can be aborted).

Chapter 5

Avoiding Interference between Goals[†]

In the previous chapter, we provided means of managing resource conflicts in an intelligent agent system. In this chapter, we investigate how goals pursued in parallel can interfere negatively with each other in terms of the outcomes of each step of the goals, and conditions that need to be maintained during certain intervals of goal pursuit. We provide data structures and algorithms that identify and avoid potential interference between goals.

When multiple goals are pursued in parallel by an agent, the effects of one goal might interfere with another goal by undoing two types of conditions:
a) *in-conditions* of an active goal or plan; or
b) *preparatory-effects* of a plan that have been achieved in anticipation of a later step, for which they are pre-conditions.

Recall that *in-conditions* are conditions that must remain true during the pursuit of a goal or execution of a plan. In order for a goal/plan to begin execution, the in-conditions (if any) must be true, and must not be undone until the goal/plan completes, as this would cause the goal/plan to fail.

In pursuing their goals, intelligent agents often use a series of plan steps where the effect of an earlier step achieves the pre-condition for a later step. These effects are termed as *preparatory-effects*. A rational agent should not allow preparatory-effects which have been achieved to be interfered with, until the later step for which they are pre-conditions completes.

For example, suppose a Mars rover agent has a goal to analyse soil samples at a location *A*. Amongst others, two of the steps to achieve this goal would be to *go*

[†]Part of the work presented in this chapter has been previously published (Thangarajah, Padgham & Winikoff 2003*b*).

to location A and *collect soil sample* in that order, with the effect *At A* achieved by the first step being a pre-condition to the other. In this case, having arrived at location *A*, a rational agent should not allow a plan which took the agent elsewhere to intervene, prior to the agent collecting a soil sample.

Pre-conditions that are not preparatory-effects need not be protected as there is no clear justification for protecting them. In particular, when defining the notion of *preparatory-effects* in section 5.1, only dependencies between nodes that are part of the same top-level goal are considered.

In this chapter, we focus on recognising linkages between in-conditions, effects and preparatory-effects of goals and plans of a single agent. We also look at controlling plan interleaving to ensure that an agent does not allow negative interference in the pursuit of separate parallel goals.

In section 5.1, we define the terminology that is used in this chapter. We describe the notion of interactions summaries (which are similar in concept to resource summaries presented in the previous chapter) in section 5.2, and provide algorithms for deriving them in section 5.3. Section 5.4 presents algorithms that detect and avoid interference between goals.

5.1 Preparatory Effects and Dependency Links

This section captures the linkages and interference that can occur between multiple goal and plan instances of an agent.

Typically, when developing a set of plans to achieve a goal, a developer is not able to consider all the ways in which the pursuit of other goals may interfere with that goal. There are two important ways in which parallel goals can interfere, which are captured and reasoned about in this chapter. Considering the logical outcomes of goals and plans, interference can occur when:

1. an in-condition is made false while a plan or goal is executing, causing the plan or goal to fail; or when
2. a preparatory-effect is made false before a plan or goal for which it is a pre-condition begins executing, preventing that goal or plan from being executed.

We will begin by defining some terminology that is used in the algorithms ahead, and use the illustration in figure 5.1 to assist in this process.

Preparatory-effect
Consider two plans P_1 and P_2 which are utilised to satisfy G, where P_1 is executed before P_2 (as shown in figure 5.1). If P_1 brings about an effect e which is a pre-condition of plan P_2, then e is termed a *preparatory-effect* (*p-effect*).

Dependent-plan
The plan that is dependent on the preparatory-effect, in this case P_2, is termed the *dependent-plan*.

Dependency-planlink
The link between the preparatory-effect (e) and its dependent-plan (P_2) is termed a *dependency-planlink*, $e \rightarrow \{P_2\}$.

Dependent-subgoal
In the event that an effect of a plan serves as a pre-condition to all the plans that satisfy a sub-goal (as is the case with effect e and sub-goal SG_3), then there is a link between the p-effect (e) and the *dependent-subgoal* (SG_3).

Dependency-subgoallink
The link between the preparatory-effect (e) and a corresponding dependent-subgoal (SG_3) is termed a *dependency-subgoallink*, $e \rightarrow \{SG_3\}$.

Dependency-link
We use the term *dependency-link* as a general term to refer to a link that is either a dependent-planlink or a dependent-subgoallink.

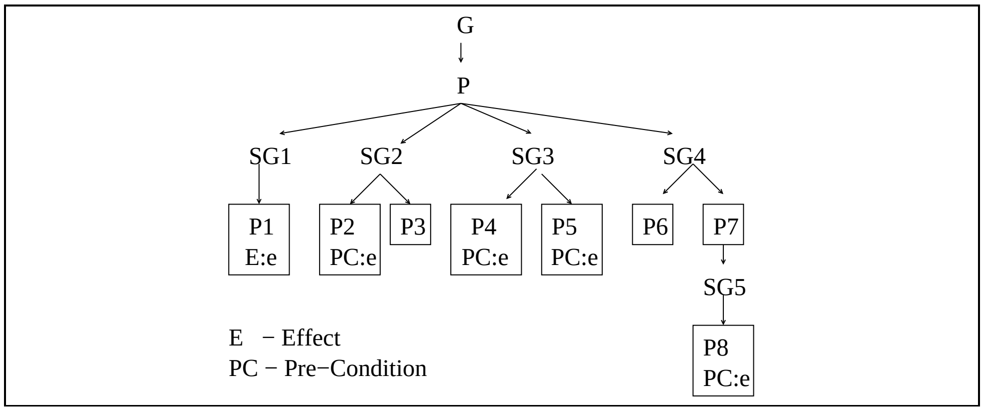

Figure 5.1: *An example of dependency-links.*

The conditions under which the different forms of dependency-links become *complete*, that is when the link no longer exists, are as follows:

- A dependency-planlink is complete when the dependent plan begins execution.

- A dependency-subgoallink is complete when either:

  a) the dependent-subgoal is complete; or

  b) the last possible plan to satisfy the sub-goal begins execution (that is, all other plans have been tried but failed).

When a p-effect is achieved, the agent should protect the p-effect from being undone until all the dependency-links associated with the p-effect are complete.

All dependency-planlinks are *potential* by definition because the agent has other means of achieving the dependent-plan's parent sub-goal.

A dependency-subgoallink can either be:

- *definite* - if the dependent-subgoal will definitely be pursued as is the case with SG_3 (figure 5.1); or

- *potential* - if the dependent-subgoal is not definite. SG_5 is an example of such a sub-goal as it depends on the path chosen for SG_4.

The above categorisations leads to a ranking of importance with respect to guarding a p-effect, in the decreasing rank of: definite dependency-subgoallink, potential dependency-subgoallink, dependency-planlink.

A definite dependency-subgoallink is of highest importance because the dependent-subgoal will definitely be pursued, and every way of achieving it require the p-effect as a pre-condition.

A potential dependency-subgoallink is more important than a dependency-planlink because, although they are both potential, the dependent-subgoal of a dependency-subgoallink will definitely fail if the p-effect is not true. In contrast, the parent sub-goal of the dependent-plan of a dependency-planlink by definition has other means of being achieved.

This ranking is useful for defining agents that are more or less careful with respect to protecting preparatory-effects from interference.

Dependency-entry:
In order to reason about and protect p-effects, dependency information is attached to each plan-node of the goal-plan tree of a particular goal. This is done at compile time at the point that the goal-plan tree is created. The dependency information maintained at each node is termed a *dependency-entry* and it consists of a set of dependency-links, and expressed as:

Dependency-entry $\longrightarrow$ *(p-effect, dependent-plan/subgoal)**

where the p-effect is an effect of the plan that is also a preparatory-effect. Dependency-entries are only attached to plans because p-effects are achieved by plans, goals only declare what the plans will achieve. An example of this is illustrated in figure 5.2.

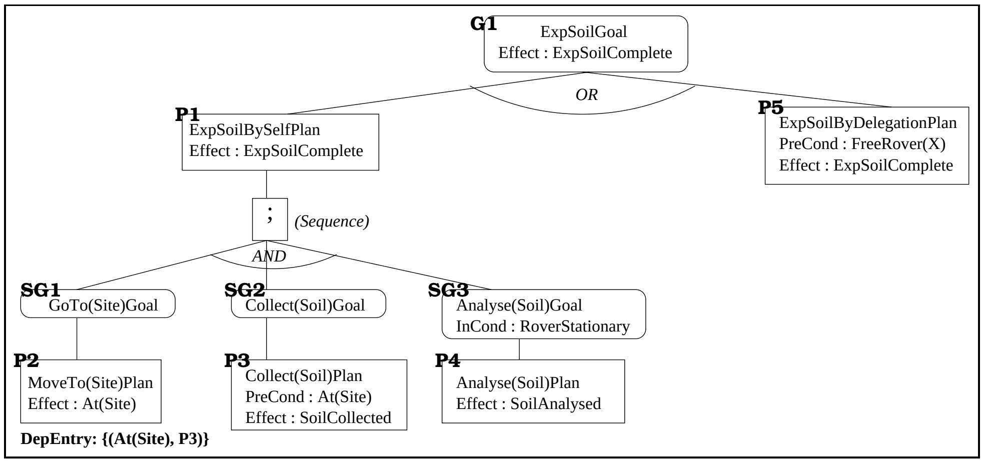

Figure 5.2: *An example of a dependency-entry.*

Algorithm for deriving dependency entries

```
procedure SetDependencyEntries(ResourceTreeNode node)
    for each child ∈ node.Children
        SetDependencyEntries (child)
    if node.Type == PlanNode then
        for each effect ∈ node.Effects
            node.dependencyEntries.add(searchUpTree(node, effect))

function searchUpTree(ResourceTreeNode node, Effect effect) returns List of dependency entries
    let dependencyList := empty list of dependency entries
    let parent := parent node of node
    if parent.Type == SequenceOperator then
        for each child ∈ parent.Children ∧ child occurs after node
        // that is each child of parent that is to the the right of node
            dependencyList.add( searchDownTree(child, effect) )
    dependencyList.add( SearchUpTree(parent, effect) )

    return dependencyList

function searchDownTree(ResourceTreeNode node, Effect effect) returns List of dependency entries
    let dependencyList := empty list of dependency entries
```

```
if node.Type == GoalNode // sub-goal
    if all the child nodes of node (i.e. the alternate plans)
    contains effect as an effect then
        dependencyList.add( new Dependency(effect, node)
if node.Type == PlanNode then
    if the node (plan) contains effect as an effect then
        dependencyList.add( new Dependency(effect, node) )

for each child $\in$ node.Children
    dependencyList.add( SearchDownTree(child, effect) )
    dependencyList.add( SearchUpTree(child, effect) )

return dependencyList
```

5.2 Interaction Summaries

In order to detect interference, we use a similar notion of summary information as used in the previous chapter to detect resource conflicts. Given that interference is caused when the effects of one goal interfere with active in-conditions and dependency-links of other goals, the summary information we maintain should summarise the following information about goals and plans:

a) effects (as they could cause the interference);

b) in-conditions; and

c) preparatory-effects (as these need to be protected when active).

Note that only those pre-conditions that are set up by p-effects are protected so there is no need to maintain pre-condition summaries.

For the discussion ahead, the general term *condition* refers to either in-conditions or effects (including p-effects). Recall that, there may be more than one way (plan) of achieving a goal. Conditions therefore, can be defined to be either *definite* or *potential*.

- A **definite condition** will *definitely* be achieved/required (achieved for effects and required for in-conditions) at some point along every possible path of achieving the goal (i.e. this condition need not be required by all plans, but is required by at least one plan in every possible way of achieving the goal). For example, the effect *ExpSoilComplete* is a *definite effect* of the goal G_1.

- A **potential condition** will *possibly* be achieved/required at some point in the pursuit of the goal. This condition is part of at least one plan in at least one path of achieving the goal, but is **not** achieved/required in all possible paths of achieving the goal. For example, the in-condition *RoverStationary* is a *potential in-condition* of the goal G_1.

Unlike resource summaries (see section 4.1), where necessary resources were a subset of the possible resources, definite and potential conditions are defined to be *exclusive* of each other.

Interaction Summary:
An *interaction-summary* of a goal/plan contains a summary of the definite and potential in-conditions, effects and p-effects of that goal/plan. The interaction-summary of a goal/plan n, $I_{summary}(n)$, is a 3 tuple, expressed as:

$$I_{summary}(n) \ = \ \langle S_i(n), S_e(n), S_{pe}(n) \rangle$$

where S_i, S_e and S_{pe} are the summaries of in-conditions, effects and preparatory-effects respectively. The in-conditions summary contains the definite and potential in-conditions of the node,

$$S_i(n) = \langle D_i, P_i \rangle$$

and each of D_i and P_i are simply sets of conditions,

$$D_i = \{ic_1, ic_2, \ldots, ic_n\} \qquad .$$

Similarly for the effects and p-effects summaries. Section 5.3 will describe how these interaction summaries can be derived.

Interaction-Tree:
We will use the term *interaction-tree* to refer to a goal-plan tree where attached at each node is the interaction-summary, and additionally for plan-nodes, the dependency-entry[1] of the plan. This tree is built at compile time for each goal-type and instantiated at run-time for each goal-instance. Figure 5.3 shows the interaction-tree of a simple goal of a Mars rover agent to analyse a soil sample.

The interaction-tree for any active goal is updated dynamically as the goal is pursued. When a plan or sub-goal completes, the corresponding node is removed and the interaction summary of the parent-node is re-computed. Any changes to the parent node are propagated up the tree. Dependency-entries are updated when plans (dependent-plans) begin execution.

[1] as described at the end of section 5.1.

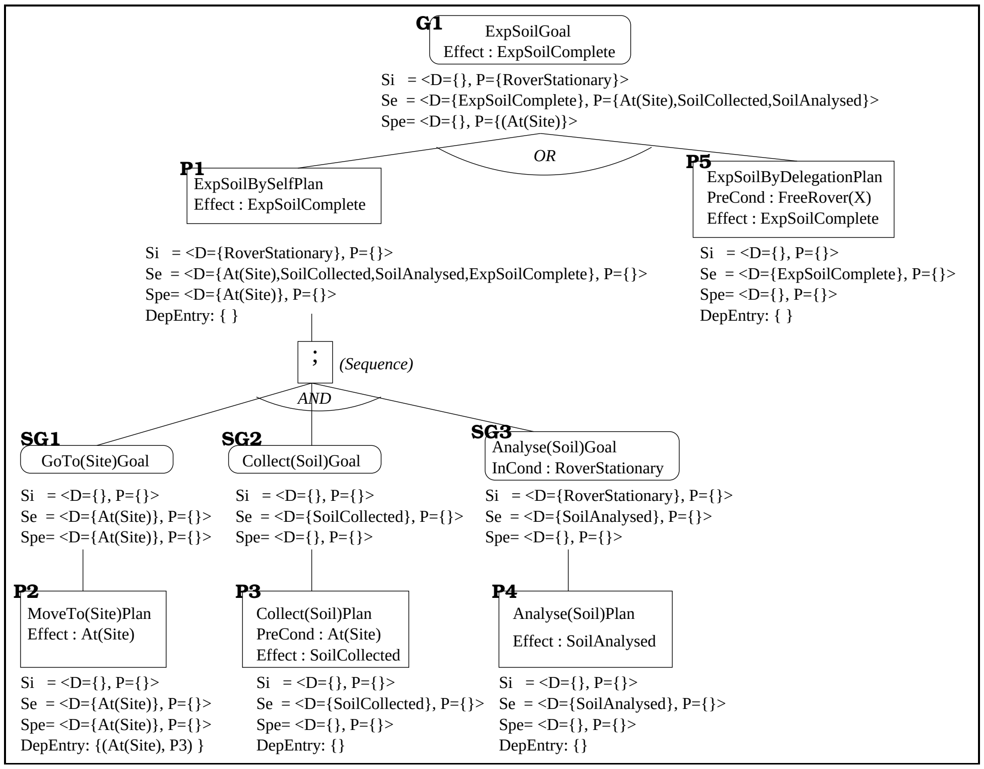

Figure 5.3: *An example of an interaction-tree.*

5.3 Deriving Interaction Summaries

Interaction summaries are derived at compile time and updated at runtime, in a similar manner to the mechanisms described in the previous chapter with respect to resource summaries (see section 4.3).

The interaction summary of a node of a goal-plan tree is derived by combining the local conditions of that node with the interaction-summaries of all of its child nodes. Each piece of the summary (i.e. the in-condition summary, the effect summary and the p-effect summary) is calculated by combining the relevant pieces of the local and child nodes. The manner in which the interaction summaries of goal nodes and plan nodes are derived are different to each other and the details are as follows:

Interaction summaries of a plan

Firstly, we look at how the *effect summary* of a plan is derived. The effect summary of a plan is computed by taking the union of the local effects of the plan with the effects of each sub-goal within the plan. The summaries of two (sub)goals are combined as follows:

The definite effects of the two (sub)goals are combined by performing a union on the definite effects of each goal. This holds true for potential effects as well. However, where an effect is both a definite effect of the first goal (i.e. in D_{E1}) and a potential effect of the second goal (i.e. in P_{E2}), then that effect is a definite effect of the resulting summary. Hence, such an effect would not also be considered potential (recall that definite and potential effects are mutually exclusive by definition).

The addition operator $\oplus$ that performs the above operation on (sub)goals, is defined as follows:

The $\oplus$ operator

$$\begin{aligned} S_e(G_1) \oplus S_e(G_2) &= \langle D_{E_1}, P_{E_1} \rangle \oplus \langle D_{E_2}, P_{E_2} \rangle \\ &= \langle\ (D_{E_1} \cup D_{E_2}), ((P_{E_1} \cup P_{E_2}) - (D_{E_1} \cup D_{E_2}))\ \rangle \end{aligned}$$

For example,

$$\begin{aligned} \langle \{At(Site)\}, \{SoilAnalysed\} \rangle \oplus \langle \{SoilCollected\}, \{At(Site)\} \rangle = \\ \langle \{At(Site), SoilCollected\}, \{SoilAnalysed\} \rangle \end{aligned}$$

The effects summary of a plan can then be defined as:

$$S_e(P) = \langle \textit{effects-of}(P), \{\} \rangle \ \oplus \ \bigoplus_{g \in sub-goals-of(P)} S_e(g)$$

where *sub-goals-of*(P) is the set of sub-goals of plan P and *effects-of*(P) is the set of effects of P. The *p-effect summaries* and *in-condition summaries* can be similarly defined:

$$S_i(P) = \langle \textit{in-conditions-of}(P), \{\} \rangle \ \oplus \ \bigoplus_{g \in sub-goals-of(P)} S_i(g)$$

$$S_{pe}(P) = \langle \textit{p-effects-of}(P), \{\} \rangle \ \oplus \ \bigoplus_{g \in sub-goals-of(P)} S_{pe}(g)$$

where *in-conditions-of*(P) and *p-effects-of*(P) are the set of in-conditions and p-effects of P respectively.

It is important to note that unlike the calculation of resource summaries, there is no need to differentiate between sequential and parallel sub-goals of a plan. This is because all sub-goals will be achieved if the plan runs successfully, therefore leading to all conditions being required/achieved.

Interaction summaries of a goal

The derivation of the interaction summary of a goal differs from a plan because for a goal the summaries of its plans are combined and these plans are *alternate* ways to achieve the goal. An agent will typically only execute one of these alternative plans[2]. Hence, the combined definite conditions are those that are definite for all plans, while the combined potential conditions are the union of the potential sets for each plan, plus those that are definite for some plan, but not for all. We define the $\otimes$ operator, which enables the above operation, as follows:

The $\otimes$ operator.

$$\begin{aligned} S_e(P_1) \otimes S_e(P_2) &= \langle D_{E_1}, P_{E_1} \rangle \otimes \langle D_{E_2}, P_{E_2} \rangle \\ &= \langle\ (D_{E_1} \cap D_{E_2}), (((D_{E_1} \cup D_{E_2}) - (D_{E_1} \cap D_{E_2})) \cup P_{E_1} \cup P_{E_2}) \rangle \end{aligned}$$

For example,

$$\begin{aligned} &\langle \{At(Site), ExpSoilComplete\}, \{\} \rangle \otimes \langle \{ExpSoilComplete\}, \{SoilAnalysed\} \rangle \\ &\qquad = \ \langle \{ExpSoilComplete\}, \{SoilAnalysed, At(Site)\} \rangle \end{aligned}$$

All the effects of a goal are also specified in its plans, therefore there is no need to combine the local effects of the goal to its plans when deriving the *effect summary* of a goal. The effect summary (S_e) of a goal G ($S_e(G)$) is defined as:

$$S_e(G) = \bigotimes\nolimits_{p \in plans-of(G)} S_e(p)$$

where *plans-of*(G) is the set of plans that could achieve the goal G. The *p-effect summary* is defined in the same manner:

$$S_{pe}(G) = \bigotimes\nolimits_{p \in plans-of(G)} S_{pe}(p)$$

The *in-condition summary* of a goal is defined as the merged in-conditions of its plan nodes, combined with the local in-conditions of the goal as follows:

$$S_i(G) = \langle \textit{in-conditions-of}(G), \{\} \rangle \ \oplus \ \bigotimes\nolimits_{p \in plans-of(G)} S_i(p)$$

Figure 5.3 (page 90) illustrates a complete example of derived interaction summaries.

[2]More than one plan may be executed in the case of plan failure. This does not change the rationale for the combination operators.

Interaction summaries of a set of goals

In order to determine the interaction-summary (IS) of a set of goals, the $\uplus$ operator is defined, which is applied to all goals in the set as follows:

$$IS(GoalSet) = \biguplus_{g \in GoalSet} IS(g)$$

where

$$IS(g_1) \uplus IS(g_2) = \langle S_i(g_1) \oplus S_i(g_2), S_e(g_1) \oplus S_e(g_2), S_{pe}(g_1) \oplus S_{pe}(g_2) \rangle.$$

5.3.1 Dynamic Updates of the Interaction-Tree

The interaction tree template for each goal-type is built at compile time and a copy is instantiated for every goal-instance at run-time. The copy of the interaction tree is updated dynamically as the agent executes its goals and plans. This provides the agent with up to date information enabling it to make more informed rational decisions.

The algorithms for performing the update is similar to that for updating resource summaries, as presented in the previous chapter in section 4.5. A summary of the algorithm is as follows:

The tree is updated at runtime by deleting plan and goal nodes as their corresponding plans and goals complete, and recalculating the summaries each time a node is deleted.

When a *plan completes* execution, either by successful completion or failure,

- the corresponding node is removed from the tree;
- if the plan has successfully completed, then the parent goal-node is also complete and hence removed from the tree, and the change is propagated up the tree;
- if the plan has failed and the parent goal has no other means of success (i.e. if all other alternate plans have also failed, or if there is no other alternate plan) then the parent goal fails, which in turn fails its parent plan. All three failed nodes are removed and the change is propagated up the tree; and
- if the plan has failed, but the parent goal has other plans (children) to attempt, then the effect summary of the parent node is recomputed, and this change is propagated up the tree.

When a *sub-goal completes* execution:

- the corresponding node is removed from the tree; and

- the effect summary of the parent node is re-computed and propagated up the tree.

Note that the change propagation is a recursive procedure.

5.4 Executing Goals Simultaneously

The previous section presented mechanisms for deriving interaction summaries. In this section we will use the derived summary information to detect and avoid negative interference.

If two goals can be identified as definitely non-interfering with each other, then they can be pursued in parallel without any monitoring or scheduling of plans. Goals that may potentially interfere with each other can be monitored to avoid any conflict, whilst being pursued simultaneously.

Goals that definitely do not interfere can be identified as follows: As discussed in section 5.1, when two goals are executed in parallel, the following two conditions are tracked and maintained:

(**a**) the in-conditions of each goal and its active plan instances must not be violated; and

(**b**) the preparatory-effects (p-effects) that have been achieved must not be undone until the dependency-links become complete.

It can be determined that a new goal, G_{new}, will not interfere with an existing set of goals, $\mathcal{G}$, if all (i.e. both definite and potential) effects of G_{new} are compatible with all derived in-conditions and p-effects of $\mathcal{G}$.

Two conditions[3] are compatible if it is possible for them to be simultaneously true. The details depend on the specific representation of conditions.

Algorithm for detecting definitely safe goals

```
function noInterference ($G_{new}$, $\mathcal{G}$) returns {true | false}

    combinedEffects_new := $D_e(G_{new}) \cup P_e(G_{new})$
    conditionsToProtect := $D_i(\mathcal{G}) \cup P_i(\mathcal{G}) \cup D_{pe}(\mathcal{G}) \cup P_{pe}(\mathcal{G})$

    for each effect $\in$ combinedEffects
        for each condition $\in$ conditionsToProtect
```

[3]More generally, sets of conditions.

```
            **if not** *compatible*(*effect*, *condition*) **then**
                  **return** false
    **return** true
```

```
**function** compatible (Condition $c_1$, Condition $c_2$) **returns** {true | false}
    Returns true if $c_1$ and $c_2$ are simultaneously possible, false if not. This will
    depend on the nature of the conditions and how they are represented.
```

In this thesis, we represent a condition as a (name,value) pair. For example, (Data-Transmitted, Yes), (AtLocation, A). If two conditions have the same name but different values, then they are not compatible.

Identifying definitely non-conflicting goals are useful because some applications may have goals that should not be suspended (as a result of scheduling), such as a goal to re-fuel, if a Mars rover is low on energy. Thus, before the agents begins pursuing such goals, the *noInterference* method defined above can be used to ascertain if they are safe from any interference with existing goals. If there is no interference, the goal is pursued. If there is interference, then either the goal is not pursued[4] or existing goals that interfere with it are pre-empted, depending on the agent's preference of goals.

5.4.1 Monitoring and scheduling goals

Interference between two goals can be avoided by pursuing the goals in sequence. However, this is too restrictive because only some steps of the goals may conflict, the others could be freely executed in parallel. A more reasonable restriction when the interference is potential but not definite would be to pursue the goals in parallel, but monitor for steps which cause conflict, and schedule them so that the new interfering steps wait until the steps that they interfere with are complete. This is done by guarding (protecting) the in-conditions and dependency-links that are active when adopting new goals and executing new plans.

For the purpose of avoiding interference, in addition to the interaction-tree structures of goals, a further data structure is requires which is termed as the *GuardedSet*[5](GS).

The Guarded-Set includes in-conditions and dependency-links that are currently active and should be guarded by the agent. Dependency-links are obtained from the dependency-entry of the associated plan-node from the interaction-tree.

[4]Goals that are not immediately pursued may be attempted later if practical.

[5]This is in fact a pair of multi-sets.

In-conditions are placed in the guarded set along with the name of the associated goal or plan instance. The Guarded-Set can be defined by the following simple grammar:

Guarded-Set $\rightarrow$ *dependency-entry** *InCondEntry**
dependency-entry $\rightarrow$ **{p-effect** *PGName***}**
InCondEntry $\rightarrow$ **{in-condition** *PGName***}**
PGName $\rightarrow$ **planInstanceName** | **goalInstanceName**

Recall that interference between goals is avoided by protecting: (i) the in-conditions while a plan or goal is executing, and (ii) the p-effects that are achieved by a plan from the effects of new plans, until the relevant dependency-links become complete. However, in doing this, care must be taken to avoid deadlocks. For example, consider the following two goals:
G_1.inCondition(x), $G_1 \rightarrow .. \rightarrow$subgoal1.effect(NOT y), and
G_2.inCondition(y), $G_2 \rightarrow .. \rightarrow$ subgoal2.effect(NOT x).
Assume that an agent is pursuing G_1, but has not yet reached *subgoal1*. The GS(Guarded-Set) would contain x, as it is an in-condition of G_1. In considering G_2, if the agent checks only the conditions in the GS, there appears to be no problem and G_2 could be pursued, placing y also in the GS. However, when *subgoal1* and *subgoal2* are respectively reached, they would then be suspended as they affect y and x respectively (both of which are in the GS), thus creating a deadlock. Consequently, before adopting a goal or plan, the agent should look ahead for effects that can be achieved by existing goals using their interaction summaries. This is to ensure that they do not interfere with the in-conditions of the new goal or plan to be executed.

Algorithm for monitoring to avoid interference

Interference between goals can be avoided by incorporating the following steps into the agent execution cycle:

1. When the agent **begins the pursuit of a goal**:
   - the in-conditions of the goal (i.e. the local in-conditions of the goal, not the derived in-conditions) are added to the *Guarded-Set*.
2. When the agent **satisfies a goal** or if the **goal fails**:
   - any in-conditions of the goal in the *Guarded-Set* are removed; and
   - if the goal is part of a dependency-subgoallink in the *Guarded-Set*, this entry is also removed.

3. When a **plan begins execution**:

    - the in-conditions of the plan (i.e. local in-conditions) are added to the *Guarded-Set*;
    - if the plan has any p-effects then the relevant dependency-entry of each p-effect is placed in the *Guarded-Set*. This is done at the *start* of the plan, instead of the end (when the p-effect is achieved) because the plan step that achieves the effect is not known in advance, and the agent should not begin another plan that could possibly undo the p-effect while the plan is executing; and
    - if the plan completes either a dependency-planlink or a dependency-subgoallink[6] of a dependency-entry that is protected in the *Guarded-Set*, then that dependency entry is removed from the *Guarded-Set*.

4. When a **plan completes or fails** execution:

    - the in-conditions of the plan that are in the *Guarded-Set* are removed; and
    - in the event of plan failure, any dependency-entries associated with p-effects of the plan are removed from the *Guarded-Set*. As the plan has failed, the steps that were expected to occur after this plan will no longer happen.

5. Before the agent **adopts a goal**, G_{new}, it should perform the following checks to avoid deadlocks. For each goal G that has an associated condition in the *Guarded-Set*:

    - If the in-conditions of G_{new} are compatible with the **combined definite and potential effects** of G, then G_{new} can be pursued in parallel with G;
    - else, if the in-conditions of G_{new} are *not* compatible with the **definite effects** of G, then they will conflict unless G_{new} completes before G. Since we have no notion of time G_{new} should not be pursued in parallel with G, but should be re-considered when the incompatible effects of G have been achieved;
    - else, if they are *not* compatible with the **potential effects** of G, the agent can either:

[6]By being the last plan to be attempted.

(a) choose to be cautious and not allow G_{new} to execute until either the incompatible effects in G are achieved, or the goals containing the incompatible effects complete; or

(b) since potential conditions depend on the path chosen, the agent can be *optimistic* and allow G_{new} to execute hoping that the incompatible effects (and resulting goal/plan failure) will be avoided.

6. **Before the agent executes a plan**, P_{new}:

   - it should check whether the effects of P_{new} are compatible with the conditions protected in the *Guarded-Set*:
     - if they are compatible, the plan can then begin execution; or
     - if they are not compatible, the agent must then wait until they are compatible before executing the plan.
   - in order to prevent deadlocks, the in-conditions and p-effects of the plan must be checked against the derived effects of existing goals that have a guarded condition, similar to when adopting a new goal in step 5.

The above mechanism gives priority to the goal that is already being pursued as opposed to the new goal. This is nevertheless totally dependent on the priority mechanism of the application. If the new goal is more important then existing goals can be preempted. Goal priorities and mechanisms for goal pre-emption however, are beyond the scope of this thesis.

Summary

Intelligent agents often have plans involving several steps, where the results of one step may set up conditions for a later step to be successful (i.e. preparatory-effects). These steps may also have conditions that must remain true during the execution of that step (i.e. in-conditions). In this chapter, we argued that agents typically pursue multiple goals in parallel, and if intelligent or rational, should manage this parallelism in a way that avoids interference between interleaving steps.

The notion of *preparatory-effects* and *dependency-links* were defined in order to capture when an effect that is achieved by a step in a plan needs to be protected until a later plan step can be executed. Algorithms were presented that described how summary information regarding dependency links, in-conditions and effects could be computed and then used to protect both preparatory-effects (before a plan/goal executes) and in-conditions (while a plan/goal executes).

These mechanisms allow an intelligent agent to pursue multiple goals in parallel without irrationally and unnecessarily acting in a way that causes failure of some goal or plan. These mechanisms capture the kind of reasoning that is done effortlessly by humans, but is not yet evident in intelligent computer agents.

The approach in this chapter uses a form of look-ahead in order to avoid potential interference. We do not however constrain the future plan choice but allow the plan choice to be determined by the agent's context which is a characteristic of BDI systems. The mechanism allows dynamic plan choices and schedule the chosen plan execution such that interference does not occur.

Analysis and experimentation presented in chapter 7 indicates that this additional reasoning is not computationally expensive and allows an agent to satisfy more goals under conditions where there are goals which interact in this way.

Chapter 6

Exploiting Positive Goal Interaction[†]

In the previous two chapters we addressed the issue of conflicting interactions between goals, with respect to limited resource availability (chapter 4) and interference between goals (chapter 5). Goals may also interact positively. In this chapter, we investigate how multiple goals pursued in parallel can assist in achieving each other.

Following is an example where two goals interact positively. Consider a Mars rover agent that has a goal of analysing soil samples at location A and a goal of analysing rock samples also at location A. The agent could pursue these goals sequentially by going to A, analysing a soil sample, returning back to the lander, then once again going to A, analysing a rock sample and returning back to the lander. Alternatively it could go to A, analyse a soil sample, analyse a rock sample and then return to the lander. Clearly the second approach is more intelligent, and is brought about by the agent reasoning about the common steps between the two goals and deducing if these common steps can be combined (merged) for the two goals. While this reasoning is not necessary for the agent to perform its tasks correctly, it can make the agent more efficient. For example, in situations where resources are limited, combining plans may assist in more goals being achieved. In the Mars rover example above, the agent will save fuel by making a single trip to location A, instead of two.

The contribution within this chapter is to provide a framework that would allow the agent to detect and facilitate such positive interactions. We use the goal-plan tree structure (see section 3.3) and extend the notion of effect summaries (see sec-

[†]Part of the work presented in this chapter has been previously published (Thangarajah, Padgham & Winikoff 2003*a*).

tion 5.2) in order to identify plans than can be merged, and define additional data structures that are used to facilitate the process of merging two or more plans. In section 6.1, we provide the extension to effect summaries, and provide algorithms for deriving them in section 6.2. The additional data structures and the algorithms for plan merging are presented in section 6.3.

6.1 Effect Summaries

In order to determine if two goals contain common steps, we maintain summarised information about the effects of each goal which is similar to the *effect summary* that we defined in the previous chapter (see section 5.2). The difference is that whilst previously we only required the summary to contain the effects, now each effect in the summary must be accompanied by a handle to the plan(s) (i.e. the instance name of each plan) that achieve the effect. So an effect entry would be a pair *(effect, planInstanceSet)*. This extension assists in scheduling plans when common effects exist.

As outlined in the previous chapter, we classify effects into *definite effects* and *potential effects*. **Definite effects** will *definitely* be met at some point along every possible path of achieving the goal[1]. **Potential effects** will *possibly* be met at some point in time during the pursuit of the goal but are not definite. In other words, definite and potential effects are exclusive of each other. For example, in figure 6.1, *DataTransmitted* is a definite effect and *CloseToLander* is a potential effect as it only occurs if *P5* is chosen.

In the previous chapter we defined the effects summary of a goal G, $S_E(G)$, as a two tuple $\langle\ D_E,\ P_E\ \rangle$ where D_E and P_E are the definite effects and potential effects of G respectively.

The data structure for definite effects (D_E) and potential effects (P_E) is now extended to include the plans that bring about each effect as follows:

$$D_E = \{(effect, \{plan, plan..\}), (effect, \{plan, ..\})..\}$$

For example, the D_E and P_E of sub-goal $TransmitResults(SG3)$ in figure 6.1 are

$D_E = \{(DataTransmitted, \{P6, P8\})\}$, $P_E = \{(CloseToLander, \{P7\})\}$

The derivation for effect summaries differs slightly in this chapter from the methods described in the previous chapter, since in the new representation the plan sets that achieve each effect are also maintained. The methods we present below calculate the effect summaries as before, but also perform extra calculations about the plans attached to each effect.

[1]Definite effects of a goal will include the primary effects of the goal, i.e. what the goal is trying to achieve, as well as side-effects of the goal that cannot be avoided.

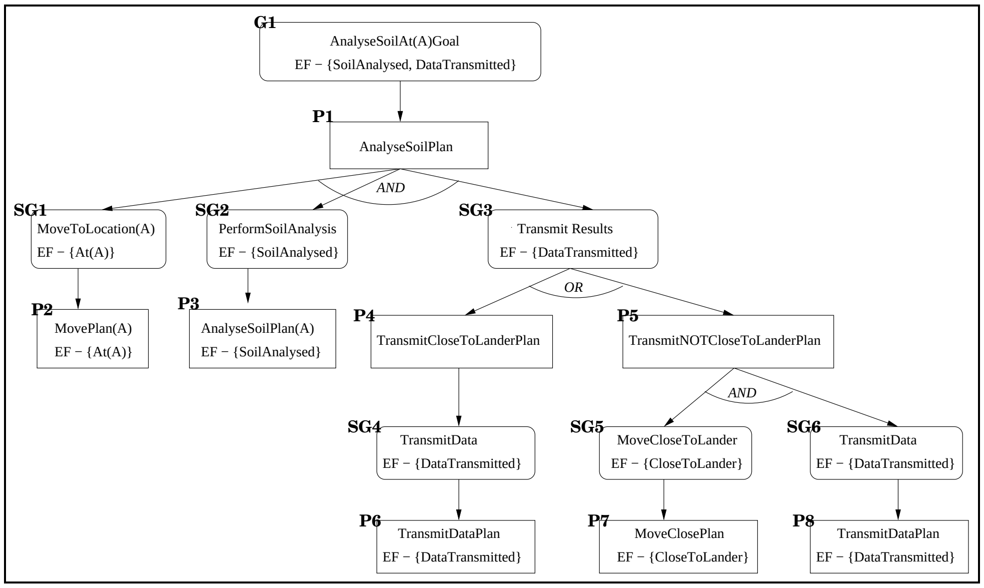

Figure 6.1: *A goal-plan tree with effects.*

6.2 Deriving Effect Summaries for Positive Interactions

We use the $\otimes$ and $\oplus$ operator that we defined in the previous chapter to combine the effect summaries of plans and goals respectively. However, given that the representation of effects are modified, we modify the $\cap$, $\cup$ and $-$, operators that are applied to a set of effects. We shall term these modified operators: $\overset{*}{\cap}$, $\overset{*}{\cup}$, and $\overset{*}{-}$, and they are defined as follows:

The $\overset{*}{\cap}$ operator

$$E_1 \overset{*}{\cap} E_2 = \{\ (e_1, p_1 \cup p_2) \mid (e_1, p_1) \in E_1 \wedge (e_2, p_2) \in E_2 \wedge e_1 = e_2\ \}$$

Remember that p_1 and p_2 here are sets of plans.

The $\overset{*}{\cup}$ operator

$$\begin{aligned} E_1 \overset{*}{\cup} E_2 \quad = \quad & \{\ (e,p) \mid (e,p) \in E_1 \wedge \neg\exists p'.(e,p') \in E_2\} \ \cup \\ & \{\ (e,p) \mid (e,p) \in E_2 \wedge \neg\exists p'.(e,p') \in E_1\} \ \cup \ (E_1 \cap^* E_2) \end{aligned}$$

where e is an effect and p is a set of plans. This definition says that $E_1 \overset{*}{\cup} E_2$ contains:

(i) any condition e that is in E_1 but not in E_2, with its associated plans;

(ii) any condition e that is in E_2 but not in E_1, with its associated plans; and

(iii) any condition e that is in $E_1 \overset{*}{\cap} E_2$, with its associated plans.

The $\overset{*}{-}$ operator

$$E_1 \overset{*}{-} E_2 = \{(e,p) \mid ((e,p) \in E_1) \wedge \neg(\exists p' \cdot (e,p') \in E_2)\}$$

Given the above definitions the $\otimes$ operator that combine the effect summaries of two plans are modified as follows:

$$\langle D_{E_1}, P_{E_1}\rangle \otimes \langle D_{E_2}, P_{E_2}\rangle \ = \ \langle \ (D_{E_1} \overset{*}{\cap} D_{E_2}) \ , \\ (P_{E_1} \overset{*}{\cup} P_{E_2} \overset{*}{\cup} ((D_{E_1} \overset{*}{\cup} D_{E_2}) \overset{*}{-} (D_{E_1} \overset{*}{\cap} D_{E_2})))\rangle$$

For example, from figure 6.2,

$$S_E(P4) \otimes S_E(P5) = \langle\{(DataTrasmitted\{P8\}), (CloseToLander\{P7\})\}\rangle \\ \otimes \langle\{(DataTransmitted\{P6\})\}, \{\}\rangle \\ = \langle\{(DataTransmitted\{P6, P8\})\}, \{(CloseToLander\{P7\})\}\rangle$$

The $\oplus$ operator which combines the summaries of two goals, is also modified as:

$$\langle D_{E_1}, P_{E_1}\rangle \oplus \langle D_{E_2}, P_{E_2}\rangle = \langle (D_{E_1} \overset{*}{\cup} D_{E_2}), ((P_{E_1} \overset{*}{\cup} P_{E_2}) \overset{*}{-} (D_{E_1} \overset{*}{\cup} D_{E_2}))\rangle$$

For example, from figure 6.2,

$$S_E(SG5) \oplus S_E(SG6) \ = \langle\{(CloseToLander\{P7\})\}, \{\}\rangle \ \oplus \\ \langle\{(DataTransmitted\{P8\})\}, \{\}\rangle \\ = \langle\{(DataTransmitted\{P8\}), (CloseToLander\{P7\})\}, \{\}\rangle$$

The effect summary of a goal G and a set of goals $\mathcal{G}$ are then derived as follows:

$$S_E(G) = \bigotimes_{p \in plans-of(G)} S_E(p)$$
$$S_E(p) = \bigoplus_{g \in sub-goals-of(p)} S_E(g) \ \oplus \ \langle\{(e, \{p.name\}) \mid e \in \textit{effects-of}(\text{p})\}, \{\}\rangle$$

$$S_E(\mathcal{G}) = \bigoplus_{g \in \mathcal{G}} S_E(g)$$

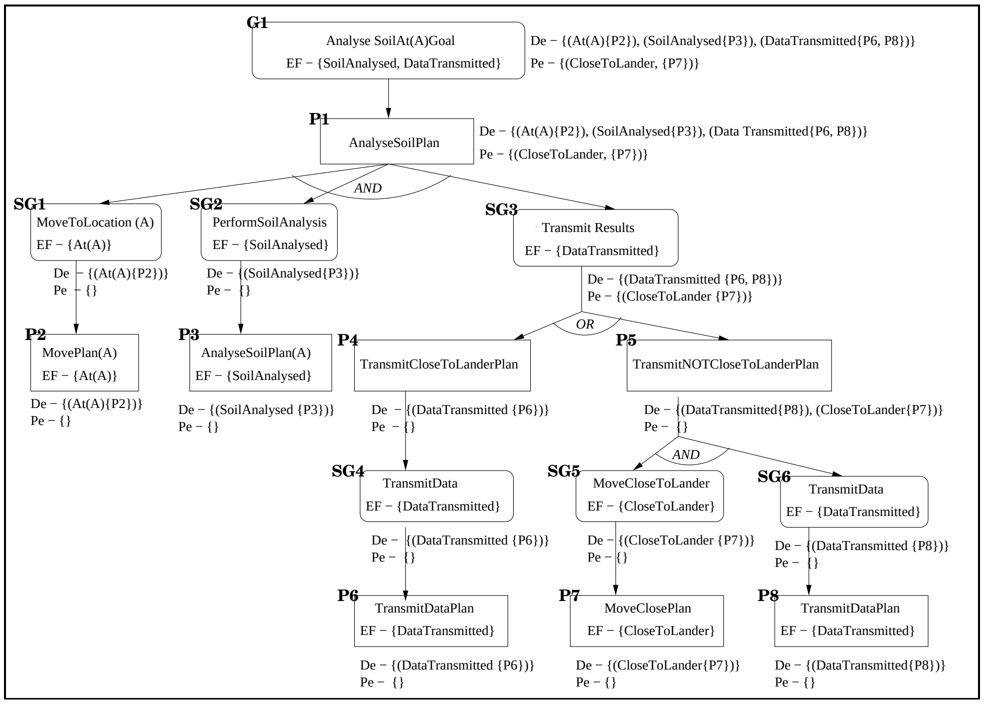
G1
Analyse SoilAt(A)Goal
EF − {SoilAnalysed, DataTransmitted}
De − {(At(A){P2}), (SoilAnalysed{P3}), (DataTransmitted{P6, P8})}
Pe − {(CloseToLander, {P7})}
P1
AnalyseSoilPlan
De − {(At(A){P2}), (SoilAnalysed{P3}), (Data Transmitted{P6, P8})}
Pe − {(CloseToLander, {P7})}
AND
SG1
MoveToLocation (A)
EF − {At(A)}
SG2
PerformSoilAnalysis
EF − {SoilAnalysed}
SG3
Transmit Results
EF − {DataTransmitted}
De − {(At(A){P2})}
Pe − {}
De − {(SoilAnalysed{P3})}
Pe − {}
De − {(DataTransmitted {P6, P8})}
Pe − {(CloseToLander {P7})}
OR
P2
MovePlan(A)
EF − {At(A)}
P3
AnalyseSoilPlan(A)
EF − {SoilAnalysed}
P4
TransmitCloseToLanderPlan
P5
TransmitNOTCloseToLanderPlan
De − {(At(A){P2})}
Pe − {}
De − {(SoilAnalysed {P3})}
Pe − {}
De − {(DataTransmitted {P6})}
Pe − {}
De − {(DataTransmitted{P8}), (CloseToLander{P7})}
Pe − {}
AND
SG4
TransmitData
EF − {DataTransmitted}
SG5
MoveCloseToLander
EF − {CloseToLander}
SG6
TransmitData
EF − {DataTransmitted}
De − {(DataTransmitted {P6})}
Pe − {}
De − {(CloseToLander {P7})}
Pe − {}
De − {(DataTransmitted {P8})}
Pe − {}
P6
TransmitDataPlan
EF − {DataTransmitted}
P7
MoveClosePlan
EF − {CloseToLander}
P8
TransmitDataPlan
EF − {DataTransmitted}
De − {(DataTransmitted {P6})}
Pe − {}
De − {(CloseToLander{P7})}
Pe − {}
De − {(DataTransmitted{P8})}
Pe − {}

Figure 6.2: *A goal-plan tree with derived effect summaries.*

Figure 6.2, shows the derived effect summaries of all the nodes of the goal-plan tree for the goal to analyse soil samples.

6.3 Identifying and Facilitating Positive Interactions

The guiding intuition here is that steps of two goals that bring about the same effect could possibly be combined (merged), hence reducing the overall cost and increasing the efficiency in goal pursuit. These steps of a goal, in BDI agents, are sub-goals. The algorithms we describe ahead, go beyond simply recognizing multiple occurences of the same sub-goal as part of different goals, and look for *any* types of sub-goals that bring about the same effects. These sub-goals are achieved by plans, and when we merge sub-goals, we are essentially merging the plans that achieve them. Some plans may bring about effects that are not intended effects of the corresponding goal (i.e. side-effects), but are intended effects of a different goal. Our algorithms also consider exploiting such situations where side-effects of a plan may be used to satisfy another goal.

We use the effect summaries to identify effects that are common to plans of different goals. Once they have been identified, we maintain a record of the plans that achieve each common effect. We differentiate between plans (with common effects) that will definitely be reached (executed) by the goals concerned, and those that may only potentially be reached; as there is a greater chance of merging plans that are definite rather than those that are potential. The structures that we use to store this information are termed the *DefinitelyMergeablePlans*(*DMP*) and the *PossiblyMergeablePlans*(*PMP*). Section 6.3.1, describes how these structures are populated.

These structures are used to monitor for when one plan that has a common effect is reached, so that it can be temporarily suspended, until the other plans, that also achieve the same common effect as the suspended plan, are reached. This process is outlined in section 6.3.2. When all the plans for a common effect are synchronized, they can be merged resulting in just one of them being executed. Section 6.3.3, discusses the conditions under which plans can be merged.

When synchronizing plans, it is possible for deadlocks to occur. For example, if a plan of goal $G1$ is suspended waiting for a plan from goal $G2$ to be reached, and a plan from $G2$ is suspended until a plan from $G1$ is reached, the cyclic dependency creates a deadlock situation. In the monitoring and scheduling algorithms we describe ahead, we have incorporated steps for preventing deadlocks. These steps rely on a data structure that we shall term the *WaitingGoalsList*(*WGL*), which maintains a list of goals that are currently suspended by the agent.

The *DMP*, *PMP* and *WGL* data structures are defined as follows:

DMP

DMP = [(effect, (goal{plan, plan, ..}), (goal{plan, ..}),..) , (effect, ...), ...]

Each entry in the *DMP* is an effect followed by the goals that bring about this effect, and associated with each goal, the set of plans of the goal that cause the effect[2]. All the effects in the *DMP* are definite effects of their respective goals in the *DMP*.

PMP

PMP = [(effect, (goal,p/d,{plan, ..}), (goal,p/d,{plan, ..})..) , (effect, ...) ..]

The *PMP* is a similar structure to the *DMP*. They differ in that whilst e is a definite effect of all its associated goals in the *DMP*, in the *PMP* e can be either a potential(p) or a definite(d) effect of the associated goals. This is explicitly represented by a flag (p/d).

WGL

[2]That is, the plans for which the effect is a direct effect.

WGL = { goalInstanceName, goalInstanceName, ... }

Note that all the information stored in the above structures can be obtained by examining the effect summaries of each goal.

We now explain how these data structures are used. To illustrate our algorithms we will refer to the sample goals illustrated in figure 6.3 below, along with their associated effect summaries as shown.

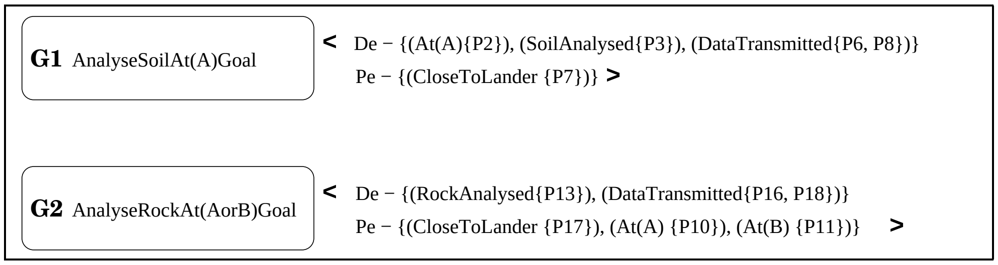
G1 AnalyseSoilAt(A)Goal
< De − {(At(A){P2}), (SoilAnalysed{P3}), (DataTransmitted{P6, P8})}
Pe − {(CloseToLander {P7})} >
G2 AnalyseRockAt(AorB)Goal
< De − {(RockAnalysed{P13}), (DataTransmitted{P16, P18})}
Pe − {(CloseToLander {P17}), (At(A) {P10}), (At(B) {P11})} >

Figure 6.3: *An example of two goals with derived effect summaries.*

6.3.1 Identifying Possible Plan Merges

When the agent decides to pursue a new goal, the effect summary of the new goal is compared with the effect summaries of current goals. If there is an effect of the new goal that is also achieved by some other goal then information is added to the *DMP* or *PMP* as follows:

- if the effect is *definite* to both goals, an entry is created and placed in the *DMP*, containing the common effect, the goals that achieve the effect and the corresponding plans of each goal that achieve the effect.
- if the shared effect is *potential* to at least one of the goals, an entry is created and placed in the *PMP*, containing the common effect, the goals that achieve the effect, a flag indicating whether the effect is definite or potential for each goal, and the corresponding plans of each goal that achieve the effect.

In both of the above cases, if there is already an entry in either the *DMP* or the *PMP*, instead of a new entry being created, the relevant information is appended to the corresponding entries.

For example, let us assume that the agent is currently only pursuing goal *G1*, and is about to begin the pursuit of *G2*. Figure 6.3, shows the up-to-date effect summaries of the two goals. When the agent adopts *G2* the *DMP* and *PMP* are populated as follows:

DMP = [(DataTransmitted, (G1{P6,P8}), (G2{P16,P18}))]

PMP = [(At(A), (G1,d,{P2}), (G2,p,{P10})),
(CloseToLander, (G1,p,{P7}), (G2,p,{17}))]

DataTransmitted is placed in the *DMP* since it is known that at least one of[3] *P6* and *P8* and at least one of *P16* and *P18* will be executed, as *DataTransmitted* is definite to the goals. Placing this information in the *DMP* indicates to the agent that there is a definite interaction between the plans of the two goals and that the goals can be scheduled to facilitate a merge of plans. However, this does not guarantee that the plans can be merged. Plan merging is described ahead in section 6.3.3.

Although *At(A)* is definite to goal *G1*, the agent cannot be certain if plan *P10* will be reached during the pursuit of *G2*, as it is only potential. Therefore, this information is stored in the *PMP*. The same is done for the effect *CloseToLander*.

6.3.2 Monitoring and Scheduling to Allow Plan Merges

In general, if we consider two goals G_1 and G_2 which the agent has just begun to pursue concurrently and a common effect, e, of the two goals, there are three different ways in which e can be common to both goals. e can be:

1 A definite effect of G_1 and G_2;

2 A definite effect of G_1 and a potential effect of G_2 or vice versa; or

3 A potential effect of G_1 and G_2.

For example, if we consider the two goals in figure 6.3,

1 - *DataTransmitted* is a definite effect of both *G1* and *G2*;

2 - *At(A)* is a definite effect of *G1* and a potential effect of *G2*; and

3 - *CloseToLander* is a potential effect of both *G1* and *G2*.

Let us now examine each case, in terms of how scheduling may be facilitated using the *DMP* and *PMP*.

Case 1 : Definite effect of both goals

[3] If a plan fails then alternatives are tried.

Effects that are common to two or more goals in this category reside in the *DMP*. The idea is to synchronize the plans by flagging them as ready to be executed when they are reached, wait for other plans that could be merged with them to be reached, and then perform a merge. This is done by incorporating the following steps that monitor the *DMP* into the agent execution cycle.

Before a plan P of a goal G is executed[4], the agent checks if P is in the *DMP*.

```
If it is in the DMP then
  let $e$ be the associated effect in the DMP
  If all the goals associated with $e$ (other than $G$) have an associated plan that
      has been flagged as ready to be executed then
      perform a merge between the plans that are ready to be executed for $e$
      and remove the entry for $e$ from the DMP.
  Else if any of the goals (other than $G$) associated with $e$, that does not have
      a plan flagged as ready (i.e. the goals that $G$ would have to wait for) is
      already in the WaitingGoalsList (WGL) then
      if no other plan associated with $e$ is flagged as ready then
          $P$ is executed and not suspended and the association of $P$ to effect
          $e$ is removed from the DMP.
      else perform a merge between the plans that are ready to be executed
          and remove the entry for $e$ from the DMP.
      While this may avoid potential merges it is necessary to avoid dead-
      locks.
  Else $P$ is flagged as ready to be executed, and the execution of $G$ is sus-
      pended. The name of the goal instance, $G$, is placed in the WGL.
```

If we consider the goals in the example shown in figure 6.3, assume that *P6* is reached in the execution of *G1*. Since it is in the *DMP*, it is flagged but not executed. The agent then waits for either *P16* or *P18* to be reached in the execution of *G2*. Let us assume that *P18* is reached. Now since all the goals that have *DataTransmitted* as a definite effect have reached a plan that brings about *DataTransmitted*, the agent would attempt to merge these plans. The process for merging the plans is described ahead in section 6.3.3.

Avoiding deadlocks

When suspending goal pursuit as done in the above algorithm, it is important to

[4]Here P and G are the instance names.

avoid deadlocks. As we have indicated we do so by using the *WGL* structure. Without this use of the *WGL*, deadlocks could occur in the following way. For example, consider the following in the *DMP*: (x, (G_a $\{p_1\}$), (G_b $\{p_2\}$)) and (y, (G_a $\{p_3\}$), (G_b $\{p_4\}$)). If p_1 is reached and G_a waits for p_2 in G_b, and then p_4 is reached and G_b waits for p_3 in G_a then there is a deadlock. In the algorithm above G_b will not be allowed to be suspended because a goal that it is waiting for (G_a) is already suspended (i.e. in the *WGL*). This method prevents deadlocks but is not the most optimal mechanism for deadlock detection and prevention. Introducing more complex deadlock detection mechanisms would increase the complexity of the agent system. If the additional complexity is acceptable then deadlock prevention mechanisms such as the resource allocation models found in operating systems (Habermann 1969) or transaction models used in distributed networks (Obermarck 1982) can be adopted. The details will not be presented in this thesis.

Case 2 : Definite effect of one goal and potential effect of another
Common effects in this category are stored in the *PMP*. This information in the *PMP* is monitored as follows:
Before an agent pursues a plan instance P of a goal G it checks if P is in the *PMP*.

If it is in the *PMP* **then**
let e be the associated effect in the *PMP*

If all the goals associated with e (other than G) have e as a definite effect **then**
move the entry for e to the *DMP* flagging P as ready to be executed.

Else (that is there is a goal associated with e (other than G) which has e as a potential effect)
Since it cannot be be certain whether the potential plan(s) of the other goal(s) will be reached, the agent has two choices:

1. Execute P and not suspend it, assuming that the potential plans will not be reached. The association of P with e is removed from the *PMP*. This approach is *cautious*, which would prevent the agent from delaying the execution of P unnecessarily if the other potential plans are never reached. However, if one of them is reached then this approach will fail to capitalise on this.
2. Flag P as ready to be executed and move this entry into the *DMP* waiting for other goals associated with e to reach a plan that achieves e. When the entry is moved to the *DMP*, although some associations are potential they are treated as definite, and managed as

described in case 1 before. This approach is *optimistic*. If the potential associations are reached then this option would attempt to exploit this interaction. If they are not reached however, P would have to wait until the other associated goals are removed from the *DMP* to begin execution. This happens when either the goals are satisfied or are no longer relevant.

In our example, if we assume that *P10* is reached first, since the remaining association to *At(A)*, *P2*, is definite, the entry is moved to the *DMP* with *P10* flagged as ready. On the other hand, if *P2* was reached first, then since *P10* is only potential, the agent can be *cautious* and execute *P2* without waiting for *P10*, or be *optimistic* and wait for *P10* to be reached.

In addition to the above monitoring, the agent may also monitor for situations where the effects in *PMP* move from being potential effects to definite effects of a goal and also when the effects are no longer applicable. This change in status can occur as plans and sub-goals of each goal complete (either by failure or success).

If effects that were potential change to be definite, and all the goals associated with the effect have it as a definite effect, then the relevant entry is moved to the *DMP*.

If the effect is no longer applicable for a particular goal, then the goal is removed from the entry associated with the effect in the *PMP*.

If there are fewer than two other goals left in the entry after this deletion, the entry is removed, because for a merge there must at least be two goals involved.

Case 3 : Potential effect of both goals
Common effects that come under this category are also placed in the *PMP*. Since in this case the effect is potential for all goals involved it cannot be guaranteed that any of the plans that achieve the effect will be reached. Hence, the agent in this situation can take one of the following options:

1. be *cautious* and execute the goals independantly.

2. be *optimistic* and place them in the *DMP*, treating them as if they were definite (case 1).

If the type of application is more resource critical than time constrained, then the optimistic approach would suit better; since it will try to merge plans in all possible situations. However, it may waste time. Therefore, if the application is more time-constrained then the cautious approach would suit better as it would ensure that the agent would not unnecessarily delay the execution of its plans for a merge that may never take place.

In the event that two goals have no common effect (neither definite nor potential) there is then no possibility of merging plans with respect to effects.

An optimization for situations with three or more goals

The mechanisms that we have defined are most suited for when two goals have a common effect. Whilst the algorithms can be applied to situations where three or more goals have a common effect, the outcomes can be inefficient for a certain case as we explain ahead.

Situations that involve three or more goals with a common effect *e*, will fall into one of the following cases:

1. all the goals have *e* as a definite effect;
2. all the goals have *e* as a potential effect (i.e. none of the goals have *e* as a definite effect);
3. only one goal has *e* as a definite effect; or
4. more than one goal (but not all) has *e* as a definite effect.

Cases 1, 2 and 3 are managed well using the algorithms that we have described previously, by placing entries into the *DMP* (for case 1) and the *PMP* (for cases 2 and 3), and scheduling appropriately.

If we follow the algorithms that we have described above for case 4, the goals will be placed in the *PMP*. This keeps the algorithms simple as the *DMP* and the *PMP* are maintained as exclusive. However, this is not desirable as plans of the goals that have e as a definite effect will definitely be reached, and by placing all of them in the *PMP*, the chance of merging these plans may be lost. For example, consider the following entry in the *PMP*:

$$(e, (G1,d,\{P1\}), (G2,d,\{P2\}), (G3,p,\{P3\}))$$

According to our algorithms, if *P1* is reached first, the agent will not wait for any merge because *P3* is potential, despite *P2* being definite. Therefore, the opportunity to merge *P1* and *P2* is lost[5].

The simplest way of dealing with this case is to ignore the goals that have *e* as a potential effect, create an entry in the *DMP* for the effect *e* with only the goals that

[5]Note that, in the same situation if *P3* is reached first, our algorithms will move the entry into the *DMP*, as the other plans are definite, which would facilitate the merge of all three plans.

have it as a definite effect, and follow the same algorithms as we have previously described. This is a sensible approach since it ensures that the opportunity for merging plans that will definitely be executed is not wasted by trying to merge plans that may never occur. Apart from being sensible, this approach also creates no additional overhead to the reasoning methods.

Whilst the above suggestion is sensible, it is not optimal; as it would miss out on situations where potential plans with common effects become definite. One way of forming an optimal solution for case 4 above would be to create an entry in the *DMP* for the goals that contain the common effect as a definite effect, as well as to create an entry in the *PMP* for all the goals concerned. This would mean that the *DMP* and *PMP* will no longer be exclusive and the algorithms will have to be modified to deal with these duplicate entries. When monitoring goal and plan execution the agent would have to first deal with entries in the *DMP*, update the *DMP* and the *PMP*, and then deal with the entries in the *PMP* making any neccesary updates. When updating the structures the agent will have to cross-check to ensure that the duplicates are handled appropriately. Although these modifications can be done, they are not trivial, and we believe that introducing additional complexity into the algorithms, that are quite simple, to monitor and merge plans that may never be reached is not justifiable. Consequently we follow the simple approach of ignoring the goals that have *e* as a potential effect and placing the others in the *DMP*.

Plans that should not be merged

The above mechanisms will identify and facilitate plan merging. However, there are instances where although plans could possibly be merged, it should not be done. For example, consider a mars rover with a goal G_1 to perform task *task1* and then *re-fuel*, and a goal G_2 to perform task *task2* and then *re-fuel*. Although these two goals have a common plan to *re-fuel* (which maps to the sub-goal of *re-fuel*), this must be performed once for each goal, because if it performs *task1* and does not *re-fuel*, it may not have sufficient fuel to complete *task2*.

In order to prevent such plans from being merged, we introduce a boolean attribute *mergeable* into the representation of the plan-type discussed in section 3.1. This attribute is assigned a value of *true* or *false* by the programmer.

6.3.3 Plan Merging

When the agent decides to merge two plans it needs to perform reasoning about the feasibility (i.e. is it possible) and choice of merge (i.e. which plan to execute).

Consider two plans P_1 and P_2 that the agent wants to merge:

- **if** P_1 and P_2 are of the same type (e.g. they are both *GoToLanderPlan*), **then** either of the plans can be executed.

- **if** P_1 and P_2 are not the same type however, they achieve only e and no other effect, **then**
either of the plans can be executed.

- **if** P_1 achieves e and x where x is a *potential effect* of the goal G_1 (i.e. side effect), and P_2 only achieves e, **then**
either of the plans can be executed since x is not a necessary effect.

- **if** P_1 achieves e and x where x is a *a definite effect* of the goal G_1, and P_2 only achieves e, **then**
P_1 must be executed because P_2 will not bring about x which is necessary for G_1.

- **if** P_1 achieves e and x, and P_2 achieves e and y where x is a definite effect of G_1, y is a definite effect of G_2, and $x \neq y$, **then**
it is not possible to merge these plans because if only one of P_1 or P_2 is executed then one of x or y will not be achieved and this is not rational as they are definite effects of the goals. In such a situation, which we will term a *useless-wait*, the plans must be executed individualy. Section 6.3.4 deals with ways of minimising the occurence of this situation.

Everytime the above merge is performed the *DMP* is modified as follows:

- The entry in question is removed from the *DMP*; and

- When the plan (in the case where a merge is performed) or plans (in the case where the plans were executed independantly) complete, the effect-summary of the associated goals are re-computed as discussed in section 5.3.1. These updated summaries are used to re-check for positive interaction between current (active) goals with respect to the effect e. The reason for re-checking *only* for the effect e is because other entries in the *DMP* are not affected by the merge. The reason for re-checking for interactions with respect to e is because there may be another pair of plans (one from each goal) that could be merged for the same effect.

6.3.4 Avoiding Useless Wait

The *useless-wait* situation described in the last section can cause the agent to be inefficient. However, under certain assumptions, this situation is unlikely to occur. If we assume that plans are cohesive, that is they are written to achieve a single effect rather than a conjunction of effects, then the useless-wait condition will not arise. Nevertheless, if required the following methods can be applied to avoid a useless-wait.

- The first approach is to apply the following filter prior to adding goals to the *DMP*. If **all** goals in question, have **at least one** plan with an intended effect (i.e. definite effect of the goal) that is not common to all the goals, these goals are then not added to the *DMP*. This is because the set of goals described can cause useless-wait.

  Whilst this method prevents the problem it is too strict and will disallow a lot of potential merges.

- The second approach is to filter goals as follows: If **all** the plans in **every** goal have a definite effect that is not common to all the goals, these goals are not added to the *DMP*. This approach will avoid pursuing goals that will definitely result in a useless-wait. However, this does not guarantee that the situation will never arise.

- A better method would be to apply the second filter, and from the goals that pass through, detect those that will never bring about an useless-wait (i.e. goals that pass through the first filter) and place them in the *DMP*. The other goals are placed in the *PMP* and each time one of them is updated with respect to interactions, the process is repeated on the set of goals in the *PMP* by applying the two filters.

  So the basic idea is that, those that definitely result in a useless-wait are not considered to be merged, those that will never result in a useless-wait will be moved to the *DMP* and the others will remain in the *PMP* and monitored further. We use this approach in the prototype system that we develop and evaluate in chapter 7.

Summary

An intelligent agent should seek not only to avoid goals that conflict but also look to identify goals that may assist in the achievement of each other. In the work presented in this chapter we focussed on the positive interactions of goals.

In this chapter, we extended the notion of effect summaries that we first introduced in the previous chapter. We also modified our algorithms for deriving effect summaries to suit the extended representation. In addition to effect summaries we introduced additional data structures which were used in the algorithms. The algorithms that we developed identified future situations where plans could be merged, and facilitated the merging of plans by monitoring and synchronising the execution of goals and plans.

In contrast to existing work on combining plans in the planning domain (Horty & Pollack 2001, Foulser et al. 1992, Nau et al. 1990), our work does not require complete knowledge of the plans to be executed in advance, which makes it suitable for intelligent agents in dynamic domains. Our data structures are constructed at compile time, which are then updated dynamically as the agent pursues goals. Therefore, the agent always reasons about goals in its current situation. The dynamic information about partially achieved goals also allows us to avoid an explicit notion of time. The representations and algorithms that are provided in this chapter are simple and efficient, and are easily implemented in agent development platforms. This is evident from our experimental work that we present in the next chapter.

Chapter 7

Evaluation[†]

In the previous chapters, we suggested mechanisms to enable *intelligent agents* built using the popular Belief Desire Intention model (BDI) (Rao & Georgeff 1992) to reason about interactions between goals that they are pursuing. Interactions may be either negative; where goals conflict, or positive; where goals support each other.

Although it is clear that more reasoning allows an agent to behave more intelligently, the cost of that reasoning is also an issue. In this chapter, we report on experiments conducted to ascertain the costs and benefits of the reasoning mechanisms about goal interactions that we have presented in the previous chapters. We find that the cost of this additional reasoning is small, even with large numbers of interactions to reason about. The benefits however, are noticeable and statistically significant, even when the amount of interactions are relatively small. Therefore, it can be concluded that adding this reasoning to agent systems is very worthwhile.

There are some similarities in the data structures and intuitions underlying the reasoning between our work and that of Clement et al. (1999*a*, 1999*b*, 1999*c*) on co-ordinating hierarchical plans in societies of agents. They too have done experimental analysis on the efficiency of their algorithms (Clement & Durfee 2001). In their evaluation, they analyse the benefits in terms of computation cost versus execution cost. They found that as more conflicts exist, the computation cost increases and the execution cost decreases (since fewer plans get executed).

In our approach, we analyse the benefits of our reasoning methods in terms of the number of goals that successfully complete, and the costs in terms of the execution and computation cost. Recall that (from section 2.6, under HTN planning), Clement et al.'s approach is somewhat different to ours in that they plan ahead and develop a plan (solution) incorporating all goals (or tasks), whilst our approach

[†]Part of the work presented in this chapter has been previously published (Thangarajah & Padgham 2004).

simply enables an agent to avoid making irrational decisions about which goals it can/should pursue. Our aims are in some ways more limited, hence we are able to realise greater efficiency in terms of computational cost.

The agent development toolkit that we have chosen to extend, incorporating the algorithms that we have presented in this thesis for the purpose of evaluating them, is JACK Intelligent Agents^{TM} (Busetta et al. 1998). JACK is a state of the art toolkit that has been used to develop a variety of large scale agent applications. In section 7.1, we describe our extension to JACK, which we will refer to as *X-JACK* (eXtended-JACK), that incorporates an explicit structure for goals and the mechanisms for managing goal interactions.

The goal achievement ability of JACK and X-JACK are then compared in a range of situations, measuring the computational cost, the run-time costs (i.e. the time taken for all the goals to complete, either successfully or by failure) and the benefits with respect to the number of goals successfully completed. Each of the three different reasoning mechanisms that were introduced (i.e. managing resource conflicts, interference and positive interactions), were individually evaluated by enabling one reasoning mechanism at a time, and disabling the others. The experimental settings and the results obtained for these evaluations on the reasoning algorithms are reported upon in sections 7.2, 7.3 and 7.4, respectively.

Hardware and software settings

All the experiments were run on an Intel Pentium 4, dedicated PC, under Linux 7.2 operating system. The speed of the processor was 1.7 GHz, and 512 MBytes of RAM were available. The scenarios were developed using JACK version 4.1 and Java version 1.4.2.

On the results obtained, when required, tests for significance (p-value tests) were done by performing the standard *Student's t-test* on the relevant data sets using *PHStat Statistical Add-In for Microsoft Excel*, version 1.4.

Where no variance was found in the data population, the p-value was determined by performing the *Wilcoxon rank sum test* which is a non-parametric equivalent of the unpaired t-test, also using PHStat.

7.1 X-JACK

We extended JACK Intelligent Agents^{TM}(Busetta et al. 1998), a Java based industrial BDI system to incorporate our data structures and reasoning algorithms about managing interactions between goals. We call this extension *X-JACK*.

There is no explicit representation of goals in JACK. Goals are implicitly represented as events which trigger the execution of plans. We extended this by introducing an explicit structure for goals, *a goal template*, that includes the attributes of *GoalTypeName*, *Plans*, *In-Conditions* and *Effects* as described in section 3.1.

A JACK plan consists of:

- a *relevance condition*, which determines if the plan is relevant to a particular event;
- a *context condition* (pre-condition), that determines the applicability of the plan given the current state of the agent;
- a *plan body*, which can be any combination of sub-goals and arbitrary Java instructions; and
- *pass* and *fail* methods, that are called when the plan succeeds or fails respectively.

In addition to the above, in X-JACK, to each plan we:

- attached the resource requirements of the plan (excluding the resource requirements of its sub-goals);
- attached the effects of the plan (excluding the effects of its sub-goals);
- attached the in-conditions of the plan (excluding the in-conditions of its sub-goals); and
- modified the relevance condition to check for relevance to a goal-type, rather than an event.

This extension to the plan structure satisfies the requirements of a plan as defined in section 3.1.

In order to manage the pursuit of goals we incorporated a *Goal Management System* (GMS) into X-JACK. Goals are added to the agent system via the GMS. At compile time, the GMS creates a goal-plan tree template for each goal-type. When a goal is added to the GMS at run-time, a goal instance is created and a goal-plan tree instance of the appropriate goal-type is attached. The GMS uses the reasoning methods developed in this thesis to identify whether the new goal is safe to be pursued with existing goals, or if monitoring is required. If the goal is safe, the GMS will allow the goal to be pursued using the normal JACK routine of plan selection and execution. If the goal is not safe, the appropriate steps as determined by our reasoning methods will be followed. When goals are safe to be pursued, the GMS will also monitor for positive interactions and facilitate any identified, using the algorithms that we have developed.

In order to facilitate the process of dynamically updating goal-plan tree instances, and for monitoring and scheduling plans, we modified:

- the plan body such that as soon as a plan is activated it registers itself with the GMS and waits until the GMS allows it to be executed; and
- the *pass* and *fail* methods of plans, such that the GMS is notified when a plan succeeds or fails respectively.

This extension also allows the GMS to monitor the progress of each goal which, although not utilised in this thesis, is a useful feature in an agent system.

7.2 Resource Conflict Management

In this section, we describe the experimental results obtained in evaluating the data structures and algorithms that we developed to reason about resource conflicts (refer chapter 4).

There is no specialised mechanism for representing resources and simulating resource usage in JACK. Therefore, we allowed resource usage to be specified at the plan levels in JACK, as we did in X-JACK. In order to simulate resource usage we introduced a *resource manager*, which we implemented identically in both JACK and X-JACK. The resource manager merely simulates resource usage and does not perform any conflict management, which was done only by the GMS in X-JACK and was not present in JACK.

The environmental variables which were controlled were: the number of goals that compete for a particular resource, the amount of resources available, the number of goals that execute in parallel at a given point in time, and the depth of the goals in terms of sub-goals and plans. These variables allowed us to both analyse the computational cost for reasoning about resource conflicts as well as to identify the kind of domains for which goal-resource related reasoning is most important.

In designing our experiments we attempted to identify situations which would make heavy use of the new data structures and algorithms in order to test them under extreme circumstances. We also identified characteristics that could be observed about applications, such as the extent to which plans tend to use the same resources, or the extent to which resources tend to be insufficient to achieve all goals. The characteristics we chose to focus on were as follows:

1. *The number of goals that run in parallel.* The more goals that run in parallel, the more they interact with each other (if they compete for resources). This allows us to test the cost of the reasoning as the number of goals increases. This characteristic also enables us to test the benefits of the system since

greater interaction also means greater chance of conflicts, which, if detected early can be avoided. We controlled the number of goals that ran in parallel by varying the delay in adding goals into the system.

2. *The extent to which goals require the same resources.* The greater the number of goals that compete for a single resource, the greater the chance for conflicts to occur. A situation with heavy use of the same few resource types provided a stress-test for the algorithms, as it increased the size of the data structures and the number of iterations within the algorithms. A high level of conflict also provided opportunities to benefit from recognition of that conflict. This variable therefore was expected to affect both costs and benefits.

3. *The amount of resources available.* Limited resource availability also contributes to greater conflict between goals, and therefore opportunities to benefit from early detection of conflict.

4. *The total number of steps that get executed for a single goal.* If a goal has many steps, there is a greater chance that it will waste resources doing early steps, before failing due to lack of resources at a later step. Thus an application with goals that require a number of steps is likely to benefit more from early conflict detection, than one where goals are accomplished in very few steps. Also, the size of the data structures grows as the number of goals and plans increase which will affect the run-time cost of the algorithms. The number of steps, or number of plans executed, arise from both the depth of the goal/plan hierarchy and the number of steps (sub-goals) within a plan.

5. *The depth of the goal/plan hierarchy.* This affects the total number of plans executed, but in addition it affects the amount of updating of resources required at run-time as the goal executes.

6. *The type of the resource.* Reasoning about conflict with re-usable resources needs to be evaluated separately from reasoning about consumable resources, as whilst conflict can be avoided by appropriately scheduling of re-usable resources, scheduling will not resolve conflicts due to consumable resources.

In order to compare the behaviour of X-JACK (incorporating our additional reasoning) with JACK, we set up abstract scenarios varying the characteristics described above. The experiments used two different structures of top level goals, as shown in figure 7.1. These are referred to as “depth 2 goals” and “depth 5 goals”. These goal structures capture both differences in plan numbers and differences in goal depth. The percentage of goals vying for the same resource was varied, as was the total availability of resources. The runtime for a single goal of depth 2

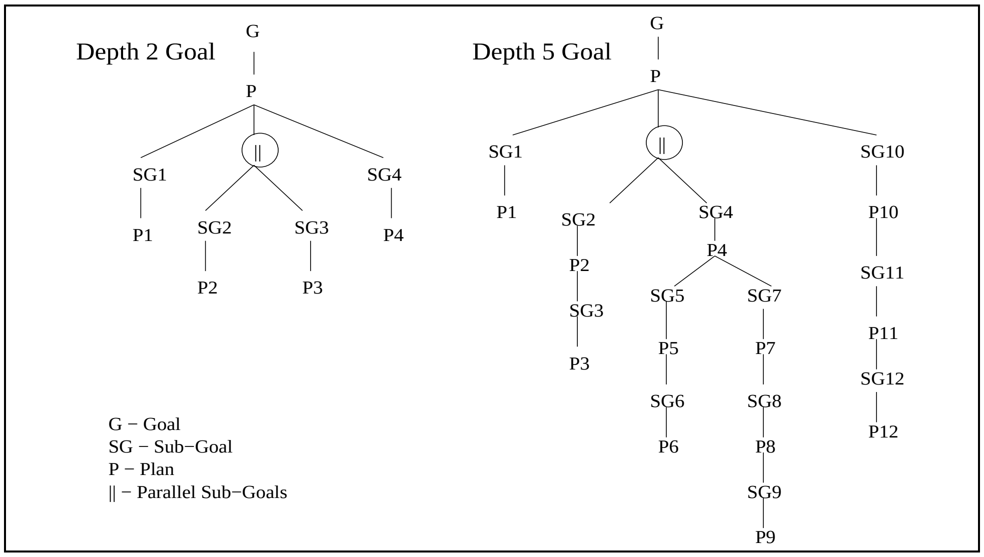

Figure 7.1: *Top level goal structures, as used in the experiments.*

was approximately 62 seconds[1], and for depth 5 was 162 seconds. The amount of parallelism was varied by changing the rates at which goals were added into the system (the shorter the interval, the greater the parallelism and vice versa).

Each experiment ran 50 goals in total, and was run 20 times, with the ordering of goals chosen randomly. In the following experiments, the time recorded was the actual clock time taken for the experiments to complete.

The experiments can be grouped into 4 different explorations:

1. Costs and benefits at differing levels of resource availability, with high levels of interaction between goals;

2. Costs and benefits at differing levels of resource availability, with normal/lower levels of interaction between goals;

3. Costs and benefits at differing levels of resource availability, with different types of resources (i.e. re-usable and consumable); and

4. Computational cost of reasoning about resource conflicts.

The run-time cost of reasoning was measured by recording the time taken for all goals to complete (either by success or failure). The benefit was measured by the number of goals that completed successfully (out of the 50 that were pursued).

[1]Artifi cial waiting was introduced into plans, to simulate real actions taking time.

High levels of goal interaction

To evaluate the algorithms in situations where there is a high level of interaction between goals, the following steps were followed:

- A scenario was set up, where each of 5 consumable resource types were used by 20% of the goals. Each goal used 2 resource types, and these were chosen to ensure high interaction - e.g. goal type 1 used resource types A and B, goal type 2 used resource types B and C, and so on.
- Resource availability was then varied to range from 25% to 100% of that required for all goals to complete.
- Separate scenarios were set up for 50 goals of depth 2 and 50 goals of depth 5 (refer to figure 7.1 for example of goal depths).
- Goals were chosen at random from the 50 available, and added to the system at a set interval. Initial experiments were run with an interval of 2 seconds, and the experiments were re-run for an interval of 20 seconds. The fastest rate of addition allowed up to 30 goals of depth 2 or 50 goals of depth 5 to have some of their execution time in parallel. The slowest rate allowed parallelism between 3 goals of depth 2 and 8 goals of depth 5.

Figures 7.2 and 7.3, show the average timings and goal completions, where goals were added every 2 seconds, and every 20 seconds respectively, with varying resource availability.

The situation with goal depth 5 and addition of goals every 2 seconds (Figure 7.2), required most processing from X-JACK, and also realised the most benefit. Under these extreme circumstances, even at 95% resource availability, there was a clear and significant benefit in using X-JACK, in terms of successful goal completions. In terms of run-time cost it can be seen that X-JACK takes more time for each run. However, it is not possible to ascertain here how much of the extra time by X-JACK was due to successfully executing more goals to completion, and how much of the extra time is spent on reasoning about conflicts. We leave this analysis to a later experiment in this section.

The situation with goal depth 2 and addition of goals every 20 seconds (figure 7.3), is the situation (in this experimental set) which offered least opportunity for benefit, but also required least additional processing. There was still a slight advantage in using X-JACK up to 75% of resource availability. This benefit was still statistically significant, with an observed level of significance (p-value) of 0.049. Differences in time taken (i.e. run-time costs) were not statistically significant at

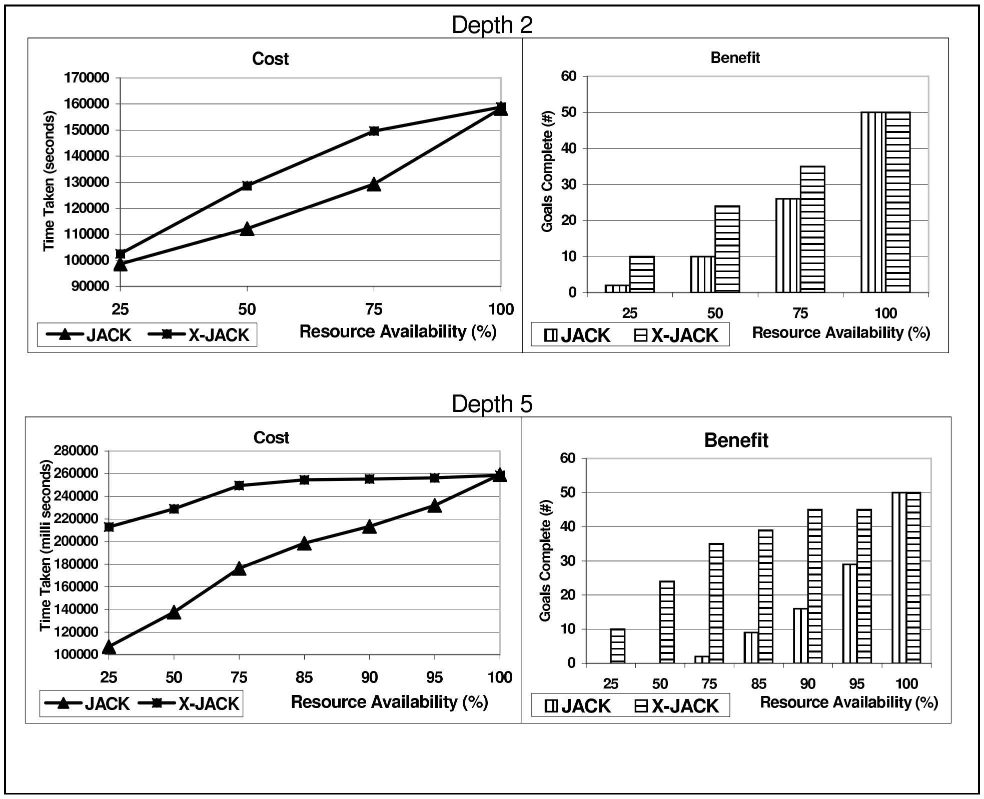

Figure 7.2: *Resource conflicts: High levels of goal interaction, with delay of 2 seconds, and high parallelism.*

the p < 0.05 level with p-values of 0.28, 0.68, 0.71 and 0.83 for resource availabilities 25%, 50%, 75% and 100% respectively.

In summary, the lower the resource levels, the higher the number of goals running in parallel and the greater the amount of plans required to complete by each goal, the greater the benefit is in using X-JACK. Even in situations of least benefit, other than where there are adequate resources to avoid conflict, X-JACK shows small yet significant benefits over JACK.

Lower/normal levels of goal interaction

In these experiments, we set up 2 scenarios (using depth 2 goals and depth 5 goals) where 40 of the 50 goals had no resource interactions (used no resources) and the other 10 goals each used 5 consumable resource types. Each of the 5 resource types was used by 20% (i.e. 2) of the goals, and each goal used 2 resource types.

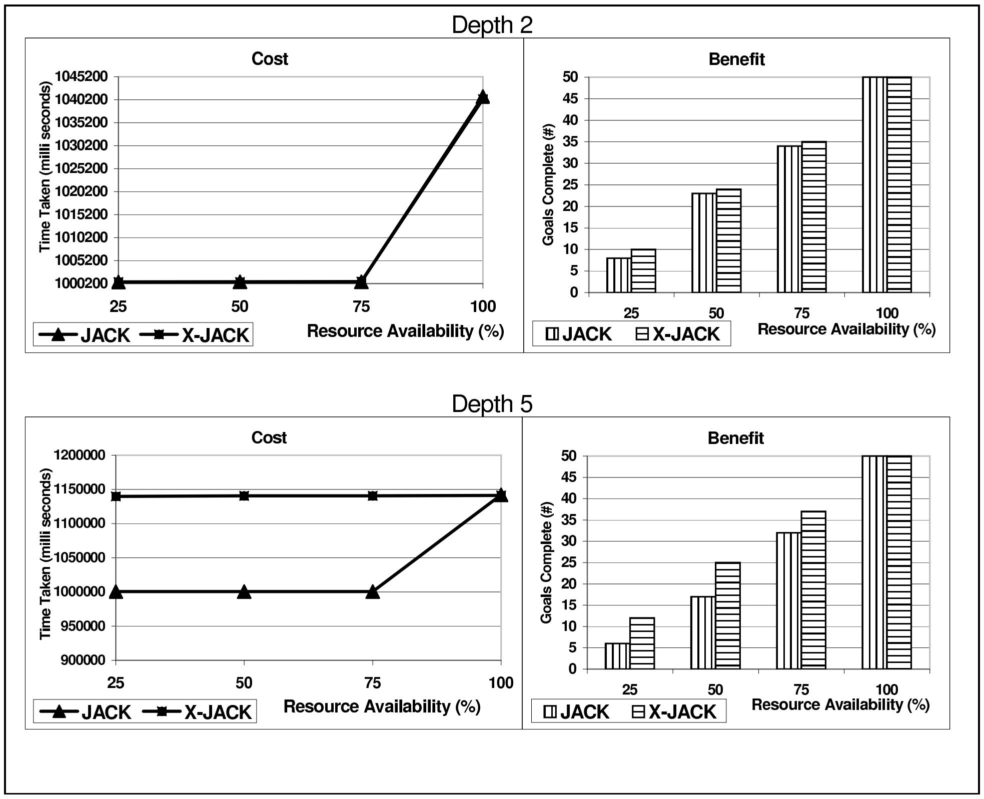

Figure 7.3: *Resource conflicts: High levels of goal interaction, with delay of 20 seconds, and low parallelism.*

This was expected to lead to a more realistic rate of conflict over resources, and therefore give a better view of potential benefits.

Figure 7.4 shows the results of these experiments. We note that even in this more realistic situation, X-JACK showed benefits in goal completions as compared to JACK, when there were insufficient resources for all goals to complete. The p-value for the difference in the number of goals that successfully completed was less than 0.00016 for 25%, 50% and 75% of resource levels, showing that the improvement in percentage of goals achieved was significant.

There was a noticeable increase in time required for X-JACK to run. However, with the exception of 50% resource availability for depth 5 (p = 0.018), differences were not statistically significant at the $p < 0.05$ level (actual values were 0.279, 0.488 and 0.285 for resource availabilities 25%, 75% and 100%, respectively).

Note again that it was not possible to determine from these experiments what proportion of the extra time was spent on reasoning and what proportion was spent

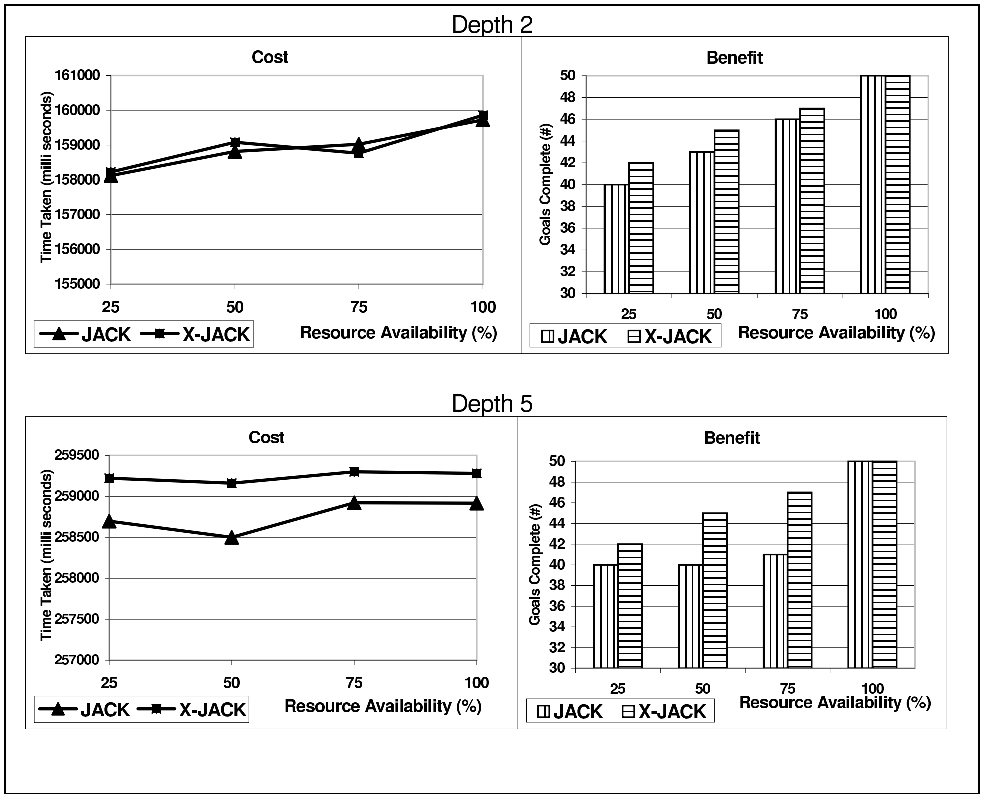

Figure 7.4: *Resource conflicts: Results with lower levels of goal interaction.*

on achieving goals which otherwise would have failed quickly.

Re-usable resources

In order to compare the behaviour of JACK and X-JACK when the resources were of type **re-usable**, we used the same experimental setup as in section 7.2, with the exception that the goals used 5 re-usable resource types instead of consumable. The results obtained are shown in figure 7.5. In this situation X-JACK completed all 50 goals irrespective of the percentage of resources available. This was because the plans which required the re-usable resources were scheduled to avoid conflict. This scheduling could however cause some run-time delay as we observed in the case of 25% resource availability for depth 5. JACK performed the same as it did for consumable resources (refer Figure 7.4).

These experiments show that while X-JACK shows benefit (in terms of goal completions) when the resources are of type consumable, the benefit is much

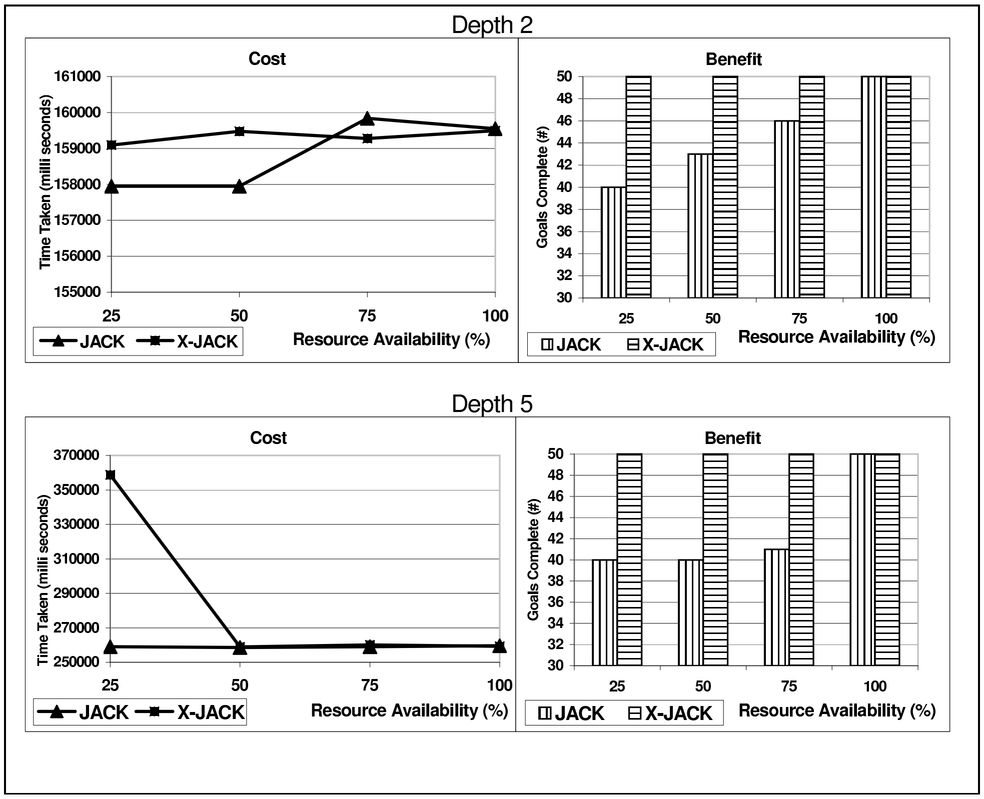

Figure 7.5: *Resource conflicts: Results with re-usable type of resources and lower levels of goal interaction.*

greater for resources of type re-usable.

Computational cost of reasoning about resource conflicts

From the above experiments, it was not possible to ascertain the actual overhead of the additional reasoning, that is, the computational costs. This is because the two systems were successfully completing a different number of goals. In order to determine the approximate cost of the reasoning, we re-ran the experiments as described for high levels of resource conflicts, for depth 5 with the goal addition interval at 2 seconds (i.e. the most stressful situation), ensuring that the same goals that completed for X-JACK also completed for JACK[2]. This gave us comparable timings for the same number of goals to run, without the additional reasoning. The

[2]This was achieved by noting the goals that was successfully pursued in X-JACK and running these same goals in JACK.

results obtained are shown in figure 7.6.

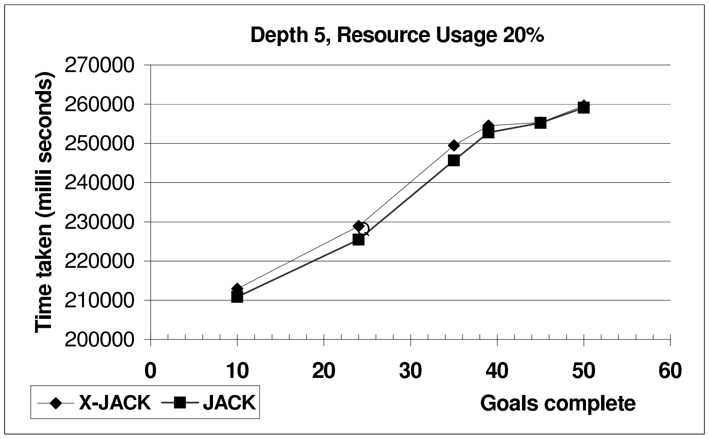

Figure 7.6: *Resource conflicts: Computational cost.*

We found that when there was no conflict (i.e. 50 goals complete), the reasoning overhead was insignificant (statistically, p-value = 0.29). Whilst there was a difference in the time taken when there were conflicts in the system, the difference was very low. For example, the situation where there were most conflicts was when 10 goals completed. The difference in the time taken between the two systems at this point was less than 1.8 seconds. Given that JACK, with the same level of conflict, did not complete any goals (refer figure 7.2), an overhead of less than 1.8 seconds to complete 10 goals as opposed to none is certainly acceptable in most situations.

7.3 Avoiding Interference

In this section, we present an experimental analysis of the costs and benefits of the reasoning algorithms and data structures that help avoid interference between goals, which we presented in chapter 5.

Similar to the experimental work on resource conflicts, we developed abstract scenarios which stress-tested the algorithms, as well as scenarios that were more realistic in real world applications.

Some of the factors that contribute towards the complexity of the algorithms that avoid interference are the amount of interaction between goals, the number of goals that execute in parallel, and the duration of the plans executed to satisfy a goal[3]. These can be explained in more detail as follows:

- *The amount of interaction between goals.* When more goals interact in terms of: (i) the effects that their plans bring about; (ii) the in-conditions of both goals and plans; and (iii) the preparatory effects set up by plans, there is a greater chance of interference. Greater interference may result in a greater number of goals failing, if the interference is not managed. High interference would also stress-test the algorithms computationally. So we expect that high interference would result in greater benefit but may also incur a higher cost.

- *The number of goals that execute in parallel.* As more goals execute in parallel, there is a greater chance of interference between goals. Therefore, there is likely to be a greater benefit from our algorithms as the amount of parallelism increases.

- *The duration of the plans required to satisfy a goal.* The longer the plans take to execute, the longer the active in-conditions and preparatory effects need to be protected, which increases the chance of interference.

Experiments were conducted to assess:

- Run-time costs and benefits of avoiding interference when there is a high level of interactions;

- Run-time costs and benefits of avoiding interference when there is a lower, more normal level of interactions;

- The effect that varying levels of parallelism have on interference between goals; and

- The computational cost of the algorithms that manage interference between goals.

In the experiments that we conducted the general setup was as follows. We created 10 different types of goals in terms of the goal/plan structure, and at run-time instantiated 20 goals whose type were randomly selected from those goal types.

We ran each experiment 50 times, and in each run selected a different random combination of goal instances. These runs were tested in JACK and X-JACK such that the 50 random combinations of goal instances were identical in both systems.

The time taken for all 20 goals to complete either by success or failure was recorded along with the number of goals that successfully completed. The former

[3]These were also factors with respect to resource conflcts as described in the previous section.

provides a run-time cost estimate while the latter indicates the benefit of using our mechanisms for avoiding interference.

High levels of interference

In order to stress-test our algorithms, we used the experimental setup as above, and set up the level of interactions such that each goal type interacted with another two goals which could potentially cause interference. This created an unrealistically high level of interactions between goals.

The results obtained in terms of the run-time cost and benefits are illustrated in figure 7.7. The figure shows that the benefit is very large; with JACK on average completing only half the amount of goals that X-JACK completes. There is however, a higher run-time cost in X-JACK, but that is expected since X-JACK executes more goals to completion than JACK, and also since when plans are suspended, they have to wait until the plans or goals that they conflict with complete.

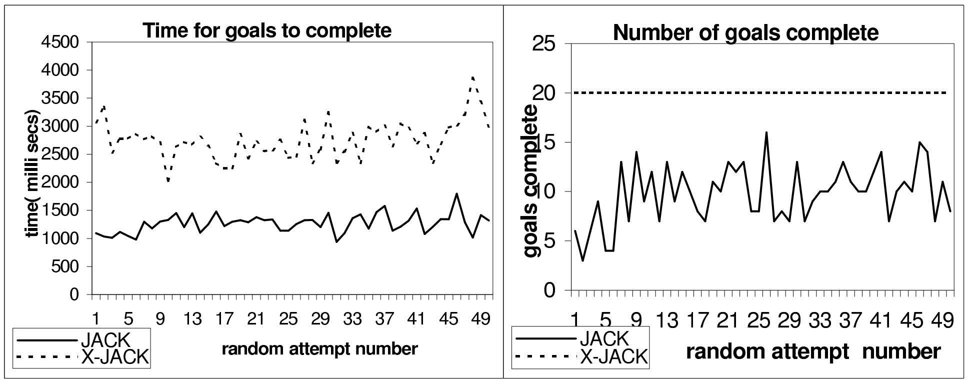

Figure 7.7: *Interference: Results with high levels of interference.*

Lower/Normal levels of interference

In order to ascertain the benefits in more realistic situations, we used the setup described above, but set the level of interference such that, 4 of the 10 goal types did not have any interactions with other goals, 3 of the goal types interacted with one other goal and the other 3 interacted with 2 other goals each.

The results obtained are shown in figure 7.8. We observe that the benefit is still quite evident, with JACK on average achieving 5 goals less than X-JACK. In terms of the run-time cost, X-JACK took significantly more time than JACK (statistically, p-value = 1×10^{-17}). This additional time may have been caused by:

- X-JACK executing more plans as it successfully completed more goals than JACK. This is not caused by the algorithms that avoid conflict, and is not a disadvantage to the system, as it is the expected run-time cost of pursuing goals.
- plans being suspended due to conflict, and waiting until it is safe to continue. This is caused by the reasoning algorithms in X-JACK. When plans are suspended in the event of *potential* conflict, the waiting time is unnecessary. However, if the plans are not suspended they may cause conflict, and as a result a lower number of goals will be achieved.
- the computational cost of the reasoning algorithms. This introduces an additional overhead into the system and it is important for the algorithms to have a low computational cost. However, we cannot ascertain this from this set of experiments and we leave it for a later set.

In order to identify the effect of having plans with longer durations, we increased the length of each plan to twice the original length[4] and re-ran the experiments. The results that we observed are shown in figure 7.9. In this case, while we do not see much of an added benefit when plans are longer, the run-time costs has increased in X-JACK. This increase was expected as goals and plans that are suspended would have to wait a longer time until the goals/plans with which they conflict completed.

Varying levels of parallelism

In this set of experiments we investigated how varying levels of parallelism between goals affect the level of interference between them. In order to do this, we used the same setup as for lower/normal levels of interactions above, but added the goals to the system via the GMS (Goal Management System) at different time intervals. The greater the time interval, the lower the chance was of the goals having steps executing in parallel.

Figure 7.10, shows the number of goals completed when the time interval of adding goals was 10, 25, 50 and a 100 milliseconds long. From the figure we can see that as the time interval is increased (i.e. parallelism decreased), the lesser the conflict that occurs, and hence the lesser the benefit of X-JACK. However, the benefit is still significant even when there are lower levels of parallelism, in that

[4]Plan lengths were increased by increasing the simulated action times within the plan.

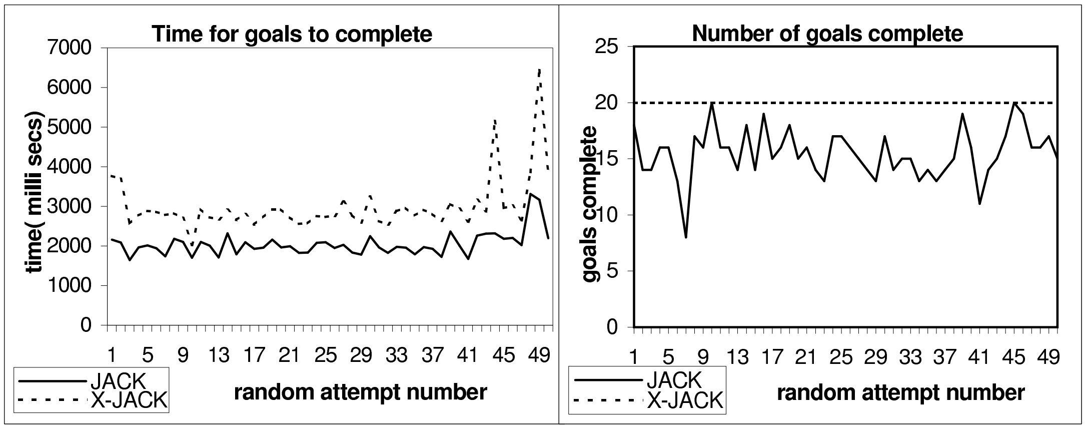

Figure 7.8: *Interference: Results with lower/normal levels of interference.*

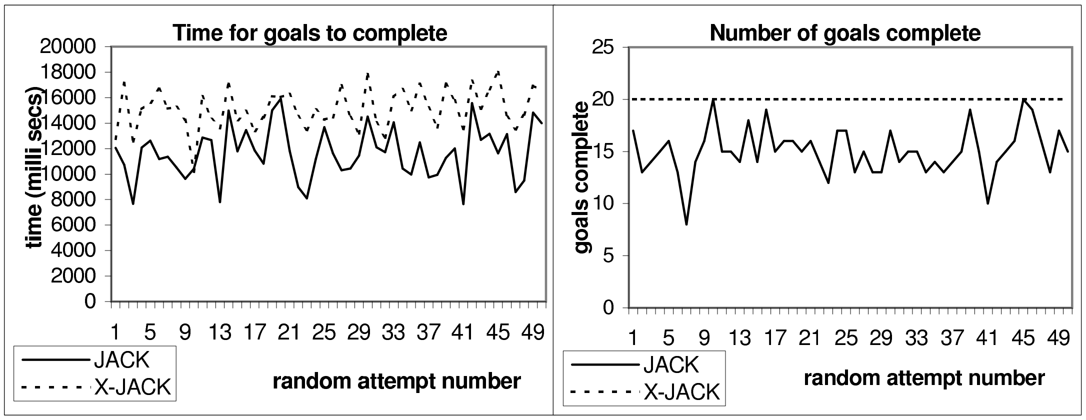

Figure 7.9: *Interference: Results with longer plans.*

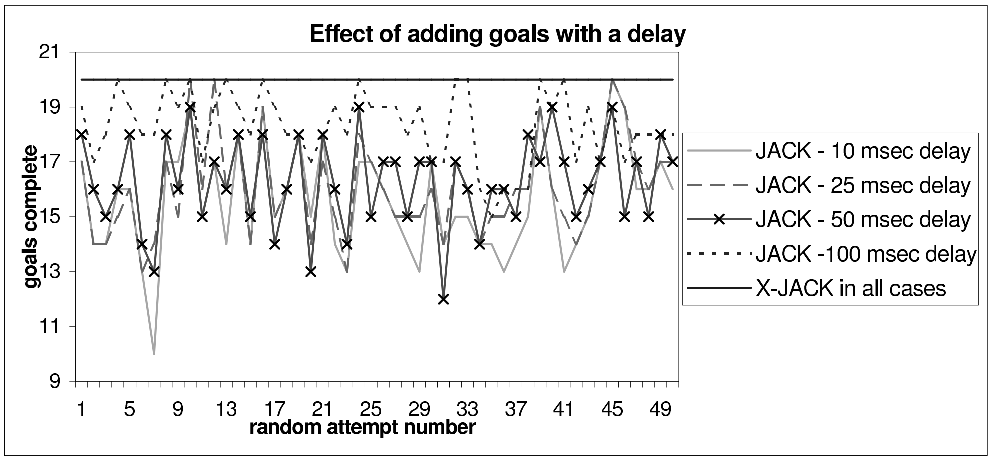

Figure 7.10: *Interference: Effect of adding goals to the system at varying time intervals.*

X-JACK always completes all the goals, but JACK only does so in 20% of the total runs.

Computational cost of avoiding interference

In order to determine the computational cost of the reasoning algorithms, we took the situation where the algorithms were stress-tested, that is the setup for high levels of interference, and modified:

- JACK, such that plans would not fail, even if the pre-conditions and in-conditions were not met[5] (that is all 20 goals complete successfully); and
- X-JACK, such that even though the reasoning algorithms were executed, plans were not suspended and were allowed to continue, and as we did for JACK, plans would not fail even if the pre-conditions and in-conditions were not met. Although the actual plan suspension was not done in X-JACK, the process of identifying interference and synchronizing plans to avoid them was still done to reflect the true computation involved.

We re-ran the experiments with the modified JACK and X-JACK and recorded the CPU usage for each run in each system. The difference in times taken to run the

[5]The GMS was modifi ed to allow all plans to be executed when chosen.

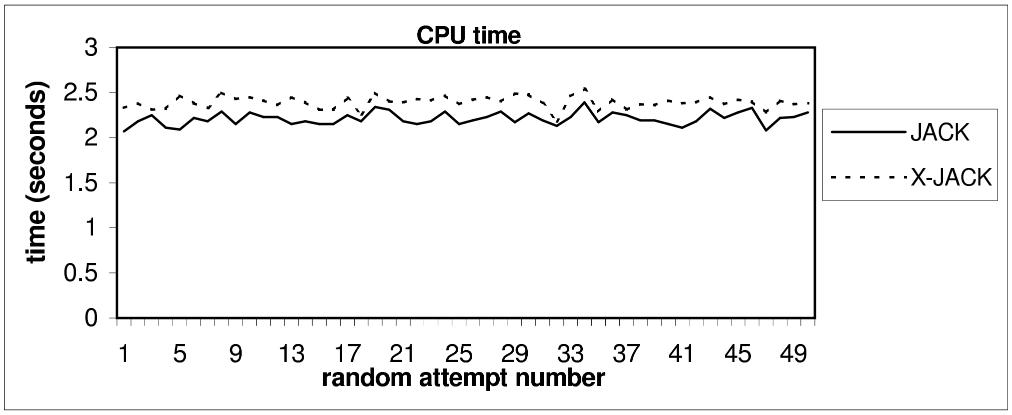

Figure 7.11: *Interference: Computational cost.*

same experiments in these two modified systems, indicates the overhead of the additional reasoning in X-JACK. The results are shown in figure 7.11.

We can see from figure 7.11, that there was a slight difference between the CPU usage of X-JACK and JACK. Statistical analysis show that this difference although small was significant. However, on average the difference over the 50 random runs is 0.18 seconds, which is a small expense for X-JACK successfully achieving twice as many goals on average than JACK (refer figure 7.7).

7.4 Facilitating Positive Interactions

In this section, we present the results of evaluating the algorithms and data structures presented in chapter 6, for detecting and facilitating positive interactions via plan merging.

The environmental variables that were controlled in these experiments were the amount of potential plan merges, the number of goals pursued in parallel, the depth of the plans/sub-goals that get merged, and the duration of the plans of the goals. These variables were chosen for the following reasons:

- *The amount of potential merges.* The greater the amount of potential merges, the more stressful it would be on the algorithms and data structures. On the other hand, this provides opportunity for greater benefit as well.
- *The number of parallel goals.* The more goals that execute steps in parallel to each other, the greater the opportunity for merging common steps, if any.

- *The depth of the plans/sub-goals that are merged.* If a plan or sub-goal that contains a lot of other sub-goals get merged (i.e. replaced by a plan of another goal), the algorithms will realise a greater benefit. For example, in figure 7.1, in the case of the goal of depth 5, there is greater benefit if plan *P4* gets merged rather than plan *P1*.
- *The duration of the plans of the goals.* Whilst this will not affect the number of plans that get merged, it will affect the run-time costs as when scheduling, a suspended plan would have to wait a longer period of time until the plan it is waiting for is reached.

Our experiments are categorized into the following explorations:

- Run-time costs and benefits of facilitating positive interactions, when there is a high level of possible plan merges;
- Run-time costs and benefits of facilitating positive interactions, when there is a lower, more normal level of possible plan merges;
- The computational cost of the algorithms that manage positive interactions; and
- The effect that varying levels of parallelism have on plan merging.

The experimental setup we used to evaluate these algorithms was identical to the setup we used to evaluate the algorithms for avoiding interference (as described in the previous section), except that instead of the goals interacting negatively, we set up the interactions such that plans between goals could possibly be merged.

To recall the setup, we created 10 different types of goals in terms of the goal/plan structure, and instantiated 20 goals whose type was randomly selected from the 10 goal types. Each experiment was run 50 times, with each run comprising 20 different random combinations of goal instances.

As we did in our earlier experiments, we measured the run-time costs by recording the time taken for all 20 goals to complete. Note that there were no negative interactions present and the goal execution is controlled such that no plans fail. Hence, all 20 goals completed in both JACK as well as X-JACK. The benefit of using X-JACK with the plan merging algorithms was that less plans needed to be executed compared to JACK to achieve the same amount of goals. Therefore, the total number of plans that get executed is also recorded to illustrate the benefits of X-JACK.

High levels of potential merges

Using the setup as above, we simulated a high level of possible plan merges as follows: Of the 10 goal types, we set up 5 goal types such that each goal type contained two plans that could possibly be merged with plans of two other different goals. The other 5 goal types contained one plan that could possibly be merged with another plan of a different goal. For each goal type, we varied the plans that could potentially be merged, with respect to their position in the goal plan tree. This setup produced an unrealistically high number of possible plan merges, but was useful in testing our algorithms in extreme situations.

The results obtained are shown in figure 7.12, from which it is evident that there is a large benefit in our reasoning methods, in terms of the number of plans executed, when there are a high number of interactions. On average, X-JACK executed 45% fewer plans than JACK. In terms of the run-time cost, X-JACK performed better as expected, since it executed a lot fewer plans (p-value = 4×10^{-32}).

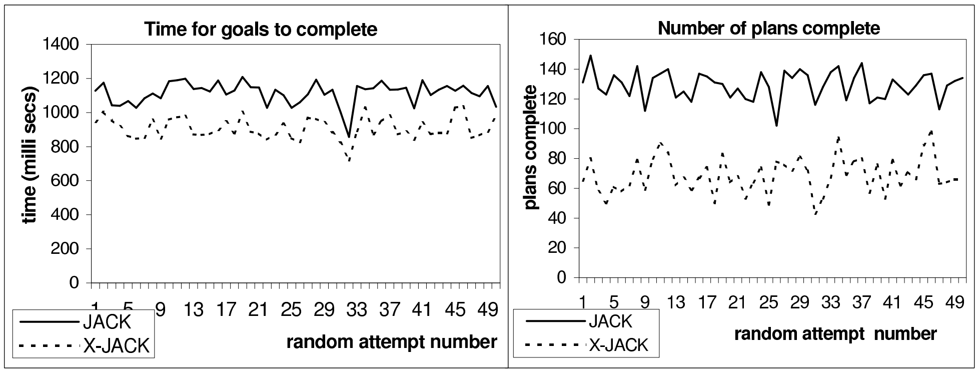

Figure 7.12: *Positive interactions: Results with high levels of possible plan merges.*

Lower/normal levels of possible plan merges.

Using the same setup as described above, we simulated a lower, more realistic level of potential merges between the 10 goal types by setting up 4 of the 10 goal types to not interact with any other goal type, 3 of the goal types to have two plans that could be merged with plans of different goals, and the other 3 goal types to have one plan that could be merged with a plan of another goal. We also ensured that

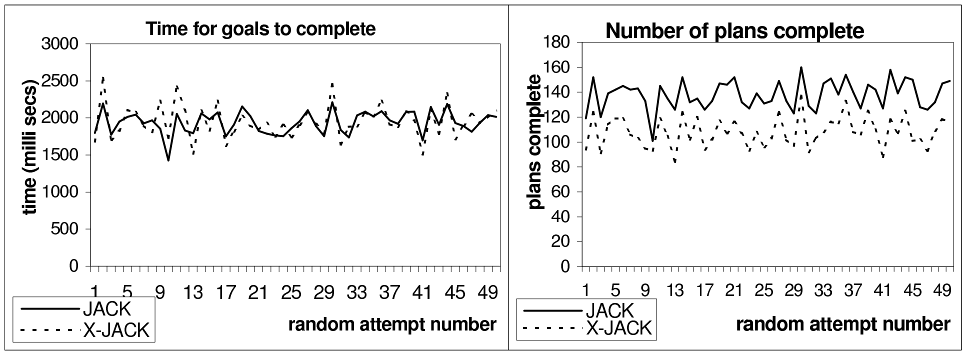

Figure 7.13: *Positive interactions: Results with lower/normal levels of possible plan merges.*

the plans that could possibly be merged were at varied positions in their respective goal-plan trees.

The results are shown in figure 7.13. We see that the benefit was still very evident; with X-JACK on average executing 29 plans less than JACK to successfully satisfy the same amount of goals. The results also showed no significant difference in the run-time costs between the two systems (p-value = 0.62). The reason for this is that although X-JACK executed a lesser number of plans, some plans were suspended in X-JACK in order to synchronise for the purpose of merging them with other plans. This incurs a run-time cost.

We then changed the duration of each plan of the goal to double its original duration, and re-ran the experiments. The results shown in figure 7.14. In this case the benefits were almost the same as before. In terms of run-time costs, although the average difference between the two systems was still statistically insignificant (p-value = 0.71), we notice that in some particular runs, X-JACK took more time than JACK, yet in other situations the converse was true. X-JACK can take more time than JACK on some occasions depending on the amount of time that a plan has to wait, until the plan that it could possibly be merged with is reached. On the other hand, X-JACK may complete faster than JACK on some occasions since there are fewer plans executed by X-JACK compared to JACK.

In order to observe the impact on the benefit of our algorithms if sub-goals of greater depth (in terms of their (sub)goal-plan tree) get merged, we re-structured the plans such that some of the plans that got merged contained such sub-goals.

The results are shown in figure 7.15. From the figure, it is very clear that there was a much greater benefit in terms of the number of plans that get executed in X-JACK over JACK; on average X-JACK executed half (50%) of the amount of plans

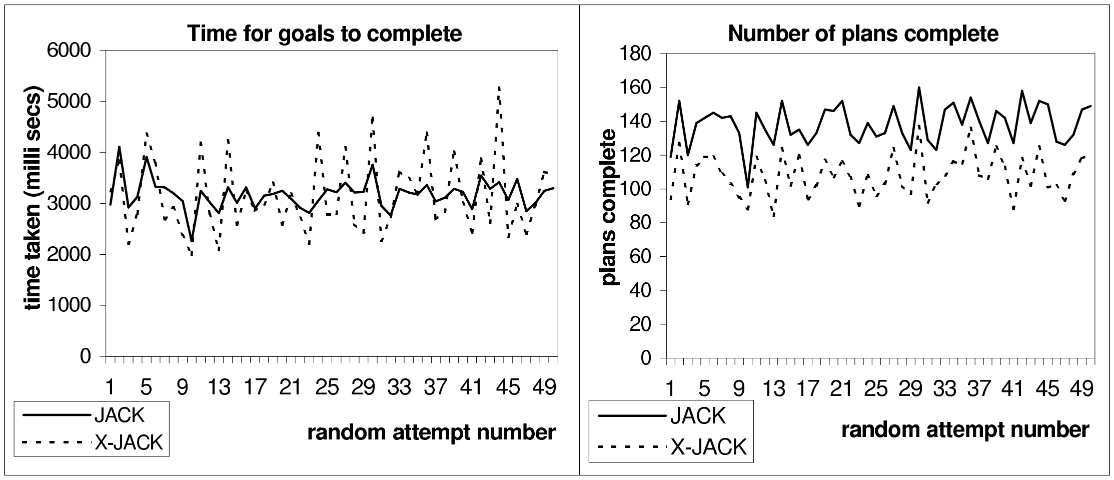

Figure 7.14: *Positive interactions: Results with longer plans.*

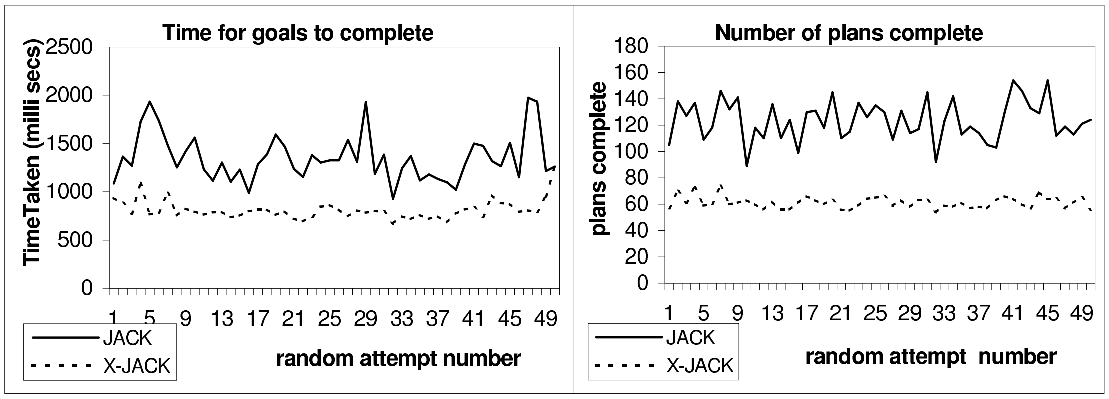

Figure 7.15: *Positive interactions: Situation where sub-goals of greater depth get merged.*

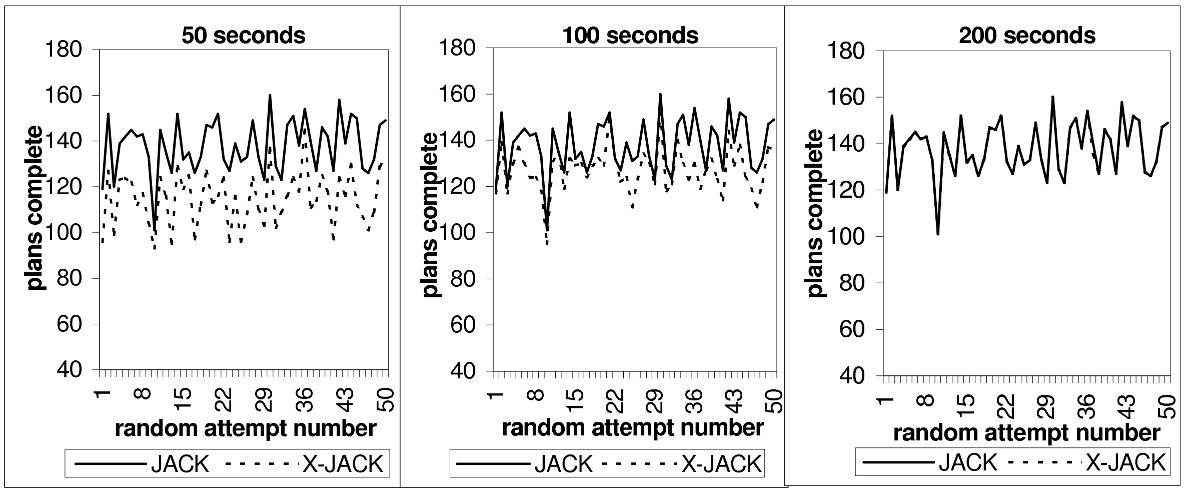

Figure 7.16: *Positive interactions: Effect of adding goals to the system at varying time intervals.*

that JACK did. It is also evident that the run-time costs of X-JACK was much less than JACK, on average X-JACK took 40% less time than JACK did. As expected, this indicates that our algorithms are most beneficial in situations where potentially mergeable steps are time consuming.

Varying levels of parallelism

In order to determine the effect that the amount of goals that occur in parallel has on the amount of plans that get merged, we took the setup for lower/normal levels of potential merges, and added the goals into the system via the GMS at varied time intervals.

Figure 7.16, shows the results for intervals of length 50, 100 and 200 milliseconds. The graphs show that as the amount of parallelism decreases, the benefit also decreases. The difference in plans executed is still significant in the favour of X-JACK, when the interval is 100 milliseconds (statistically, p-value = 5×10^{-15}). However, when the interval is 200 milliseconds, although there are a few goal steps that occur in parallel, on average the difference in the number of plans executed is insignificant between the two systems (statistically, p-value = 0.105). Our algorithms are therefore, more beneficial when there is a higher level of parallelism between goals.

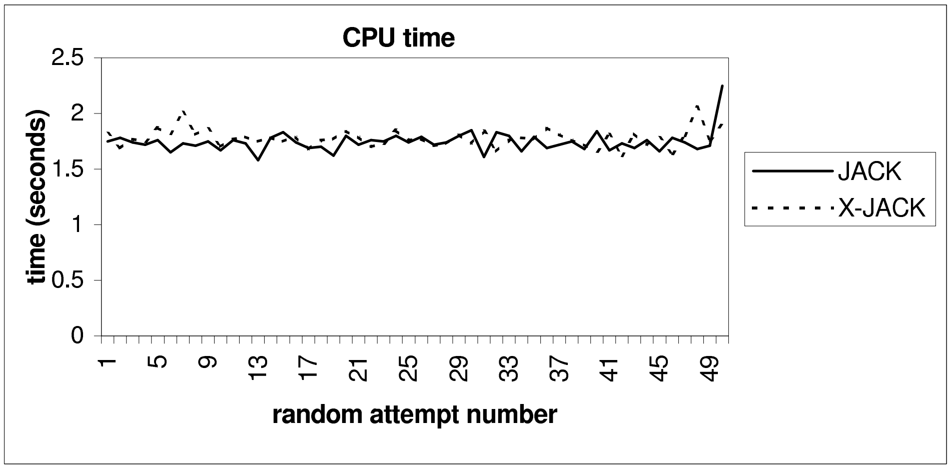

Figure 7.17: *Positive interactions: Computational cost.*

Computational cost of facilitating positive interactions

The previous experiments gave us an indication of the run-time costs but not the computational cost of our algorithms. In order to determine the computational cost, we took the setup where there were high levels of possible plan merges and made the following changes to X-JACK. X-JACK was modified such that the algorithms for identifying plan merges were executed, but the plans were not actually merged but allowed to execute normally as they would in JACK. Although the actual plan merging was not done, the process of tracking the effects, delaying plans, and determining which plans to merge was done to reflect the true computation involved.

We re-ran the experiments with JACK and with this modification to X-JACK and recorded the CPU usage for each run in each system. The difference in times obtained was the overhead of the additional reasoning in X-JACK and the results are shown in figure 7.17.

The results show that, the difference in CPU time between X-JACK and JACK over the 50 random runs was statistically insignificant with a p-value of 0.053. As there was an insignificant overhead and there may be substantial gain, it would seem to make sense to always use these algorithms. Situations where they would not be used in, are where it is important for some goals to be completed as quickly as possible, even if overall time would be saved in delaying the goals.

7.5 Conclusion

The ability to pursue multiple goals is one of the hallmarks of an intelligent agent, and it is important that this is able to be done in a rational way. In the previous chapters we have developed mechanisms for managing both positive and negative interactions between multiple concurrent goals of an agent. In this chapter we described experimental work, which evaluated the costs (both run-time and computational) and benefits of those mechanisms that we developed.

We extended JACK Intelligent Agents^{TM} to incorporate the data structures and algorithms that we have developed, and called this extended system X-JACK. In order to ascertain the costs and benefits of each of the three different kinds of reasoning we presented, about resource conflicts, interference, and positive interactions between goals, our experiments were conducted individually for each of them. In each case we described the experimental setting including the factors that contribute towards the complexity of the reasoning. We used these settings to explore the costs and benefits of X-JACK over JACK under conditions that were extreme in order to stress-test our algorithms, as well as under conditions that were more realistic.

Results clearly show that even under extreme circumstances of very high levels of goal interaction, the cost is not excessive for any of the reasoning mechanisms we developed. The benefits however, are very large in such circumstances. In situations where the costs and benefits of our reasoning mechanisms were expected to be at a minimum, while costs were insignificant (statistically at a 0.05% p-value level), the benefits were still statistically significant.

The results presented here clearly indicate that incorporating the suggested data structures and reasoning algorithms into an agent development toolkit like JACK, allows intelligent agents to be more rational and productive without sacrificing their efficiency.

Chapter 8

Conclusion and Future Work

Intelligent agents are becoming increasingly popular as a technology that can be used to develop large scale applications, in particular where the domain is highly dynamic and complex. In such applications, an agent typically pursues multiple goals concurrently. As we have shown, these goals can interact either negatively when the interactions cause one or more goals to fail, or positively when there is opportunity for one goal to assist in the achievement of another. An "intelligent" agent should therefore, consider these interactions and be rational in the way it pursues its goals.

Summary of contributions

In this thesis we have developed mechanisms that manage the interactions between goals, both negative and positive. The mechanisms were developed to be domain independent, and were based on the notion of summary information. The key idea is to summarise relevant information at compile-time, maintaining an abstract representation, which can be used to reason about goal interactions at run-time. We have shown how this notion of summary information can be adopted, in a computationally effective manner, to manage three different forms of interactions: conflicts due to resource limitations, interference due to interleaving goal-steps, and situations where the steps of one goal assists in achieving another goal. These mechanisms allow an agent to behave in a more sensible manner than that supported by current practical implementations.

In current agent development systems like JACK, in order to avoid resource conflicts in an application, the programmer would need to consider all possible situations that could lead to conflict and implement appropriate mechanisms that manage each conflicting situation. Given the many different possible ways in which the agent may choose to pursue its goals and the concurrent execution of multiple goals, this task may not always be feasible, particularly if the application is developed by many, or evolves over time. On the other hand, if the mechanisms that we have proposed are integrated into a system like JACK, then the programmer would only need to specify the resource requirements of each plan, as the plan is developed, in order to manage resource conflicts. This approach is less error-prone as the programmer has to deal with individual, modular plans rather than the entire system. Further, our approach is more friendly towards adding new plans into the system as the programmer is only required to add information that is local to the plan and does not have to consider its interaction with other plans in the system, nor make any changes to existing plans to accommodate new plans.

The potential benefits of integrating our mechanisms into an agent system are quite significant. This is shown in our experimental evaluations. The evaluations also showed that these mechanisms were not computationally expensive. This is one of the main features of our algorithms; they are simple and efficient. This is important because the agents that we develop operate in domains where they are required to deliberate and act in a timely manner. Our work therefore, enables agents to be more rational in how they pursue goals, without sacrificing their efficiency.

Another key feature of our reasoning methods is that they are not domain dependent and are easily integrated into an agent development system that follows the BDI model of agents. The mechanisms therefore, provide support for reasoning about goal interactions at an infrastructure level rather than being application specific.

A BDI agent typically has multiple ways in which it can achieve its goals (and corresponding sub-goals, if any). The path chosen depends on the context that the agent is situated in. Hence, it is not sensible for an agent to pre-determine the solution to a particular goal. However, it is possible to maintain information required to reason about goal interactions at abstract levels, denoting them as either definite (that is, they are true no matter which way the agent satisfies the goal) or potential (that is, they are true only in some ways of satisfying the goal). We identify the relevant information for each of our reasoning mechanisms, and provide algorithms for deriving this information at abstract levels and for updating this information at run-time. This ensures the agent always reasons about up-to-date information.

In chapter 4, we presented mechanisms that utilised the summarised information to identify various categories of resource conflicts. Possible solutions for each conflict category were also presented in this chapter. The solutions we have proposed are simple and do not consider, for example, goal priorities. However, once conflicts are detected between goals and a goal choice is necessary to avoid conflict, various mechanisms can be used including priority to ensure that goals with higher priority succeed.

We presented mechanisms for identifying and avoiding interference, and mechanisms for identifying and facilitating positive interactions between goals of an agent, in chapters 5 and 6 respectively.

For our experimental work, we extended JACK Intelligent Agents^{TM} to incorporate our reasoning methods. We found that integrating our reasoning mechanisms into JACK was straight-forward. We called the extension X-JACK and evaluated it against JACK. We reported on the experimental settings and the outcomes in chapter 7.

Limitations

Although generic, the mechanisms that we have developed cannot be simply applied to any arbitrary agent application. Apart from following the Belief Desire Intention (BDI) model of agents, the application must be developed with some limitations in order for our mechanisms to be useful. For example, in terms of plan and sub-goal structure, the sub-goals within a plan must be either sequential, parallel or a nested combination of the two. The mechanisms do not consider other forms of sub-goals, such as conditional sub-goals (that is, those that are required only if some condition is true) within a plan body. This can be overcome by using plan-choice to represent conditional sub-goals, where the pre-condition of the plan indicates the condition that must be true for the sub-goal to be pursued. This however, must be considered at the design stage. Further, we assume that the programmer is aware of algorithm-specific information, such as the resource requirements and effects of each plan, at compile time. This may not be possible for some applications. Also, we do not allow this pre-specified information to be changed dynamically at run-time. Therefore, after the application has been deployed no further changes can be made to such information without re-compiling the application.

Future work

The type of goals that we focussed on, and built our reasoning methods for, are achievement goals. An achievement goal is a type of goal that represents a state of the world that the agent would like to achieve. Once achieved, the goal is dropped. Some of the algorithms that we have developed could be adapted to accommodate other types of goals, such as maintenance goals (where a maintenance goal is a state of the world the agent must maintain for a given period of time). For example, the reasoning about resource conflicts does not depend on the type of the goals but on the resource usage of their plans.

However, other reasoning methods that we have developed are not directly applicable to other goal types. For example, in managing interference between goals, we protect the active in-conditions and preparatory effects of a goal. If the goal was a maintenance goal, then the agent must also maintain the state of the world as specified by the goal. Further work is required to extend our reasoning methods to accommodate other goal types.

The reasoning methods that we have developed are specifically for a single-agent with multiple goals. We used the notion of summary information for this purpose. Future work could investigate the feasibility of extending the concept of summary information along with the reasoning methods that we have developed to reason about interactions between goals of different agents that operate in the same environment. A simple extension would be for each individual agent in a multi-agent setting to communicate their summary information either to each other or to a central control agent. A key issue in this approach, would be deciding when to communicate changes to the summaries of goals. On the one hand, frequent communication would allow each agent to be working with up-to-date information, whilst on the other hand, it is not desirable to send a message after every execution step. Although this extension is possible, the communication overhead may be impractical.

In our algorithms for avoiding interference (section 5.4) and facilitating positive interaction (section 6.3.1), we monitor plan execution and, when needed, suspend plans, waiting for another plan to complete or to begin execution. This monitoring and scheduling of plans could lead to deadlock situations. We incorporated rather simple, possibly sub-optimal mechanisms for avoiding deadlocks into our algorithms. Future work could also involve investigating more optimal mechanisms for avoiding deadlock situations, whilst maintaining the simplicity of our algorithms.

Our extension to JACK, X-JACK, is available for public use. The relevant binaries and documentation can be downloaded from `www.cs.rmit.edu.au/~johthan/X-JACK`. In order to use X-JACK it is necessary for both JACK version 4.1. and Java version 1.4. to be pre-installed on the client system.

Bibliography

Agre, P. E. & Chapman, D. (1987), PENGI: An Implementation of a Theory of Activity, *in* ‘Proceedings of AAAI-87’, Seattle, WA, pp. 268–272.

Alchourrón, C., Gärdenfors, P. & Makinson, D. (1985), ‘On the Logic of Theory Change: Partial Meet Functions for Contraction and Revision’, *Journal of Symbolic Logic* (50), 510–530.

Aylett, R., Brazier, F., Jennings, N. R., Luck, M., Preist, C. & Nwana, H. (1998), ‘Agent systems and applications’, *Knowledge Engineering Review* **13**(3), 303–308.

Bell, J. & Huang, Z. (1997), Dynamic goal hierarchies, *in* ‘Proceedings from the Workshop on Intelligent Agent Systems, Theoretical and Practical Issues’, Springer-Verlag, pp. 88–103.

Bellman, R. E. (1957), *Dynamic Programming*, Princeton University Press, Princeton.

Blythe, J. (1999), ‘Decision-theoretic planning’, *AI Magazine* **20**(2), 37–54.

Blythe, J. & Veloso, M. (1992), An analysis of search techniques for a totally-ordered nonlinear planner, *in* J. Hendler, ed., ‘AIPS-92: Proceedings of the First International Conference on Artificial Intelligence Planning Systems’, Kaufmann, San Mateo, CA, pp. 13–19.

Bond, A. H. & Gasser, L. (1988), An Analysis of Problems and Research in DAI, *in* A. H. Bond & L. Gasser, eds, ‘Readings in Distributed Artificial Intelligence’, Morgan Kaufmann Publishers, San Mateo, CA, pp. 3–35.

Booch, G. (1994), *Object-Oriented Analysis And Design With Application*, 2nd edn, Addison-Wesley.

Booth, R., Chopra, S., Ghose, A. K. & Meyer, T. (2003), Belief liberation (and retraction), *in* ‘Proceedings of the Ninth Conference on Theoretical Aspects of Rationality and Knowledge’, ACM Press, University of Indiana, Indiana, pp. 159–172.

Booth, R., Chopra, S., Meyer, T. A. & Ghose, A. K. (2004), A unifying semantics for belief change, *in* ‘Proceedings of the Sixteenth European Conference on Artificial Intelligence (ECAI-2004)’, IOS Press, Spain, pp. 793–797.

Bordini, R. H. & Hübner, J. F. (2004), Jason: A Java-based AgentSpeak interpreter used with saci for multi-agent distribution over the net, Technical report, http://jason.sourceforge.net/.

Boutilier, C. & Brafman, R. I. (1997), Planning with concurrent interacting actions, *in* ‘Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)’, AAAI Press / MIT Press, Providence, Rhode Island, pp. 720–726.

Bratman, M. (1987), *Intentions, Plans, and Practical Reason*, Harvard University Press, Cambridge, MA.

Bratman, M. (1999), *Faces of Intention: selected essays on intention and agency*, Cambridge University Press, Cambridge, UK.

Bratman, M., Israel, D. & Pollack, M. E. (1988), ‘Plans and resource-bounded practical reasoning’, *Computational Intelligence* **4**, 349–355.

Braubach, L., Pokahr, A., Moldt, D. & Lamersdorf, W. (2004), Goal representation for BDI Agent systems, *in* ‘The Second International Workshop on Programming Multi-Agent Systems (ProMAS)’, New York, USA, pp. 9–20.

Brooks, R. A. (1986), ‘A robust layered control system for a mobile robot’, *IEEE Journal of Robotics and Automation* **RA-2, No1**, 14–23.

Brooks, R. A. (1991*a*), ‘Integrated systems based on behaviours’, *SIGART Bulletin* **2**(4), 46–50.

Brooks, R. A. (1991*b*), Intelligence without reason, *in* ‘Proceedings of the Twelfth International Joint Conference on Artificial Intelligence’, Morgan Kaufmann, Sydney, Australia, pp. 569–595.

Brooks, R. A. (1991*c*), ‘Intelligence without representation’, *Artificial Intelligence* **47**(1-3), 139–159.

Busetta, P., Rönnquist, R., Hodgson, A. & Lucas, A. (1998), JACK Intelligent Agents - components for intelligent agents in Java, Technical report, Agent Oriented Software Pty. Ltd, Melbourne, Australia.

Busetta, P., Rönnquist, R., Hodgson, A. & Lucas, A. (1999), 'JACK Intelligent Agents - components for intelligent agents in Java', AgentLink News, Issue 2.

Chapman, D. & Agre, P. E. (1987), Abstract Reasoning as Emergent from Concrete Activity, *in* M. P. Georgeff & A. L. Lansky, eds, 'Reasoning about Actions and Plans', Kaufmann, Los Altos, CA, pp. 411–424.

Clarke, E. H. (1971), 'Multi-part pricing of public goods', *Public Choice* **11**, 17–33.

Clement, B. J., Barrett, A. C., Rabideau, G. R. & Durfee, E. H. (2001), Using abstraction in planning and scheduling, *in* 'Proceedings of the Sixth European Conference on Planning (ECP-01)'.

Clement, B. J. & Durfee, E. H. (1999*a*), Identifying and Resolving Conflicts among Agents with Hierarchical Plans, *in* 'Proceedings of the AAAI Workshop on Negotiation: Settling Conflicts and Identifying Opportunities, AAAI Technical Report WS-99-12', pp. 6–11.

Clement, B. J. & Durfee, E. H. (1999*b*), Theory for Coordinating Concurrent Hierarchical Planning Agents using Summary Information, *in* 'Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Innovation Applications of AI conference on Artificial Intelligence and Innovative Applications of Artificial Intelligence', American Association for Artificial Intelligence, pp. 495–502.

Clement, B. J. & Durfee, E. H. (1999*c*), Top-down search for coordinating the hierarchical plans of multiple agents, *in* 'Proceedings of the Third Annual Conference on Autonomous Agents', ACM Press, pp. 252–259.

Clement, B. J. & Durfee, E. H. (2001), Performance of Coordinating Concurrent Hierarchical Planning Agents Using Summary Information, *in* 'Proceedings of the Seventh International Workshop on Intelligent Agents VII. Agent Theories Architectures and Languages', Springer-Verlag, pp. 213–227.

Cohen, P. & Levesque, H. (1990*a*), 'Intention is choice with commitment', *Artificial Intelligence* **42**(2-3), 213–261.

Cohen, P., Levesque, H. & Smith, I. (1997), 'On team formation', *Contemporary Action Theory, Synthesis* **2**, 87–114.

Cohen, P. R. & Levesque, H. J. (1990*b*), Persistence, Intention, and Commitment, *in* P. R. Cohen, J. Morgan & M. E. Pollack, eds, 'Intentions in Communication', MIT Press, Cambridge, MA, pp. 33–69.

Conte, R. & Castelfranchi, C. (1995), Understanding the functions of norms in social groups through simulation, *in* N. Gilbert & R. Conte, eds, 'Artificial Societies: The Computer Simulation of Social Life', UCL Press: London, pp. 252–267.

Cox, J. S. & Durfee, E. H. (2003), Discovering and exploiting synergy between hierarchical planning agents., *in* 'Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS 2003)', Melbourne, Australia, pp. 281–288.

Dastani, M., de Boer, F., Dignum, F. & Meyer, J.-J. (2003), Programming agent deliberation: an approach illustrated using the 3APL language, *in* 'Proceedings of the Second International Joint Conference on Autonomous agents and Multi-agent Systems', ACM Press, Melbourne, Australia, pp. 97–104.

Dastani, M., van Riemsdijk, B., Dignum, F. & Meyer, J.-J. (2003), A programming language for cognitive agents: Goal-directed 3APL, *in* 'The First International Workshop on Programming Multi-Agent Systems (ProMAS)', Melbourne, Australia, pp. 111–130.

Dignum, F. & Conte, R. (1997), Intentional agents and goal formation, *in* 'Agent Theories, Architectures, and Languages', pp. 231–243.

d'Inverno, M., Kinny, D., Luck, M. & Wooldridge, M. (1998), A formal specification of dMARS, *in* 'Proceedings of the Fourth International Workshop on Intelligent Agents IV, Agent Theories, Architectures, and Languages', Springer-Verlag, pp. 155–176.

d'Inverno, M., Luck, M., Georgeff, M., Kinny, D. & Wooldridge, M. (2004), 'The dMARS Architecture: A Specification of the Distributed Multi-Agent Reasoning System', *Autonomous Agents and Multi-Agent Systems* **9**(1-2), 5–53.

Doyle, J., Shoham, Y. & Wellman, M. (1991), A logic of relative desire (preliminary report), *in* Z. W. Ras & M. Zemankova, eds, 'Methodologies for Intelligent Systems: Proceedings of the Sixth International Symposium ISMIS'91', Springer, Berlin, Heidelberg, pp. 16–31.

Ephrati, E. & Rosenschein, J. (1991), The Clarke Tax as a Consensus Mechanism Among Automated Agents, *in* 'Proceedings of the Ninth Conference on Artificial Intelligence', pp. 173–178.

Erol, K., Hendler, J. A. & Nau, D. S. (1994*a*), UMCP: A sound and complete procedure for hierarchical task-network planning, *in* 'Artificial Intelligence Planning Systems', pp. 249–254.

Erol, K., Hendler, J. & Nau, D. S. (1994*b*), HTN planning: Complexity and expressivity, *in* 'Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)', Vol. 2, AAAI Press/MIT Press, Seattle, Washington, USA, pp. 1123–1128.

Estlin, T., Gray, A., Mann, T., Rabideau, G., Castaño, R., Chien, S. & Mjolsness, E. (1999), An integrated system for multi-rover scientific exploration, *in* 'Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference', American Association for Artificial Intelligence, Orlando, Florida, USA, pp. 613–620.

Ferguson, I. A. (1992), TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents, PhD thesis, Computer Laboratory, University of Cambridge, Cambridge, UK.

Fikes, R. & Nilsson, N. (1971), 'STRIPS: A new approach to the application of theorem proving to problem solving', *Artificial Intelligence* **5(2)**, 189–208.

Fikes, R. & Nilsson, N. (1990), STRIPS: A new approach to the application of theorem proving to problem solving, *in* J. Allen, J. Hendler & A. Tate, eds, 'Readings in Planning', Kaufmann, San Mateo, CA, pp. 88–97.

Foulser, D. E., Li, M. & Yang, Q. (1992), 'Theory and algorithms for plan merging', *Artificial Intelligence* **57**(2-3), 143–181.

Franklin, S. & Graesser, A. (1997), Is it an agent or just a program? : A taxonomy for autonomous agents, *in* 'Proceedings of the Third International workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages', Springer, pp. 21–35.

Gärdenfors, P. (1988), *Knowledge in Flux: Modeling the Dynamics of Epistemic States*, MIT Press.

Gärdenfors, P., ed. (1992), *Belief Revision*, Cambridge Press.

Georgeff, M. P., Pell, B., Pollack, M., Tambe, M. & Wooldridge, M. (1999), The Belief-Desire-Intention Model of agency, *in* J. Müller, M. P. Singh & A. S. Rao, eds, 'Proceedings of the Fifth International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)', Vol. 1555, Springer-Verlag: Heidelberg, Germany, pp. 1–10.

Georgeff, M. P. & Rao, A. S. (1996), A profile of the Australian Artificial Intelligence Institute, *in* 'IEEE Expert', pp. 89–92.

Ghose, A. K., Hadjinian, P. O., Sattar, A., You, J.-H. & Goebel, R. G. (2004), 'Iterated Belief Change', *Computational Intelligence* **20**(1), 37–55.

Graham, J. R., Decker, K. S. & Mersic, M. (2003), 'DECAF - A Flexible Multi Agent System Architecture', *Autonomous Agents and Multi-Agent Systems* **7**(1-2), 7–27.

Grosz, B. J. & Sidner, C. L. (1990), Plans for discourse, *in* P. R. Cohen, J. Morgan & M. E. Pollack, eds, 'Intentions in Communication', MIT Press, Cambridge, MA, pp. 417–444.

Habermann, A. N. (1969), 'Prevention of system deadlocks', *Communications of the ACM* **12**(7), 373–377.

Hindriks, K., de Boer, F., van der Hoek, W. & Meyer, J. C. (2000), Agent programming with declarative goals, *in* 'Intelligent Agents VI - Proceedings of the Seventh International Workshop on Agent Theories, Architectures, and Languages (ATAL'2000)', Springer Verlag, Berlin, pp. 228–243.

Hindriks, K. V., Boer, F. S. D., Hoek, W. V. D. & Meyer, J.-J. C. (1999), 'Agent programming in 3APL', *Autonomous Agents and Multi-Agent Systems* **2**(4), 357–401.

Horty, J. F. & Pollack, M. E. (2001), 'Evaluating new options in the context of existing plans', *Artificial Intelligence* **127**(2), 199–220.

Howard, R. (1960), *Dynamic Programming and Markov Processes*, MIT Press, Cambridge, Massachusetts.

Huber, M. J. (1999), JAM: a BDI-theoretic mobile agent architecture, *in* 'Proceedings of the Third Annual Conference on Autonomous Agents', ACM Press, pp. 236–243.

Ingrand, F. F., Georgeff, M. P. & Rao, A. S. (1992), 'An architecture for real-time reasoning and system control', *IEEE Expert: Intelligent Systems and Their Applications* **7**(6), 34–44.

Ioannidis, Y. E. & Sellis, T. K. (1989), ‘Conflict resolution of rules assigning values to virtual attributes’, *SIGMOD Rec.* **18**(2), 205–214.

Jennings, N. R., Parsons, S., Noriega, P. & Sierra, C. (1998), On argumentation-based negotiation, *in* ‘Proceedings of the International Workshop on Multi-Agent Systems’, Boston, USA, pp. 36–45.

Jennings, N. R. & Wooldridge, M. (1998), Applications of intelligent agents, *in* ‘Agent Technology: Foundations, Applications, and Markets’, Springer, chapter 1, pp. 3–28.

Jung, H., Tambe, M. & Kulkarni, S. (2001), Argumentation as distributed constraint satisfaction: Applications and results, *in* ‘Proceedings of the Fifth International Conference on Autonomous Agents’, ACM Press, pp. 324–331.

Katsuno, H. & Mendelzon, A. O. (1991), On the difference between updating a knowledge base and revising it, *in* J. Allen, R. Fikes & E. Sandewall, eds, ‘Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference (KR’91)’, Kaufmann, San Mateo, CA, pp. 387–394.

Kinny, D., Ljungberg, M., Rao, A. S., Sonenberg, E., Tidhar, G. & Werner, E. (1992), Planned team activity, *in* C. Castelfranchi & E. Werner, eds, ‘Artificial Social Systems — Selected Papers from the Fourth European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW-92 (LNAI Volume 830)’, Springer-Verlag: Heidelberg, Germany, pp. 226–256.

Kobayashi, N. (2000), Type Systems for Concurrent Processes: From Deadlock-Freedom to Livelock-Freedom, Time-Boundedness, *in* ‘Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics’, Springer-Verlag, pp. 365–389.

Koehler, J. (1998), Planning under resource constraints, *in* ‘Proceeding of the Thirteenth European Conference on Artificial Intelligence’, Brighton, UK, pp. 489–493.

Kripke, S. A. (1986), Semantical Analysis of Modal Logic I: Normal Modal Propositional Calculi (gekuerzte Übersetzung), *in* K. Berka & L. Kreiser, eds, ‘Logik-Texte: Kommentierte Auswahl zur Geschichte der Modernen Logik (vierte Auflage)’, Akademie-Verlag, Berlin, pp. 177–181.

Ljungberg, M. & Lucas, A. (1992), The OASIS air-traffic management system, *in* ‘Proceedings of the Second Pacific Rim International Conference on Artificial Intelligence (PRICAI ’92)’, Seoul, Korea, pp. 236–243.

Ljungberg, M., Rao, A., Sonenberg, E., Tidhar, G. & Kinny, D. (1992), Representing and executing social plans, *in* ‘Proceedings of the Second Pacific Rim International Conference on Artificial Intelligence’, Seoul, Korea, pp. 1283–1289.

Luck, M. & d’Inverno, M. (1998), Motivated Behavior for Goal Adoption, *in* C. Zhang & D. Lukose, eds, ‘Multi-Agent Systems: Theories, Languages, and Applications, Fourth Australian Workshop on Distributed Artificial Intelligence’, Vol. 1544 of *Lecture Notes in Computer Science*, Springer, pp. 58–73.

Maes, P. (1989), ‘How to do the right thing’, *Connection Science Journal* **1**(3), 291–323.

Maes, P. (1991), ‘The Agent Network Architecture (ANA)’, *SIGART Bulletin* **2**(4), 115–120.

Maes, P. (1994), Modelling adaptive autonomous agents, *in* ‘Artificial Life’, Vol. 1, pp. 135–162.

Marsh, S. (1994), Trust in Distributed Artificial Intelligence, *in* C. Castelfranchi & E. Werner, eds, ‘Artificial Social Systems: Selected Papers from the Fourth European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW’92)’, Springer, Berlin, Heidelberg, pp. 94–112.

Matthies, L., Gat, E., Harrison, R., Wilcox, B., Volpe, R. & Litwin, T. (1995), Mars microrover navigation: performance evaluation and enhancement, *in* ‘Proceedings of the International Conference on Intelligent Robots and Systems’, Vol. 1, IEEE Computer Society, p. 433.

Meyer, T., Ghose, A. & Chopra, S. (2001), Non-Prioritised Ranked Belief Change, *in* ‘Proceedings of the Eighth Conference on Theoretical Aspects of Rationality and Knowledge’, Morgan Kaufmann Publishers Inc., pp. 151–162.

Michon, J. A. & Akyürek, A. (1992), *SOAR: A Cognitive Architecture in Perspective - A Tribute to Allen Newell*, Kluwer Academic Publishers, Boston.

Mohanty, H., Patra, M. R. & Naik, K. S. (1997), Influencing: A Strategy for Goal Adoption in BDI Agents, *in* ‘Proceedings of the Second International Conference on Cognitive Technology (CT ’97)’, IEEE Computer Society, pp. 175–184.

Müller, J. P. (1994), A conceptual model of agent interaction, *in* S. M. Deen, ed., ‘Proceedings of the Second International Working Conference on Cooperating Knowledge Based Systems (CKBS-94)’, Dake Center, University of Keele, UK, pp. 389–404.

Müller, J. P. & Pischel, M. (1994), Modelling interacting agents in dynamic environments, *in* ‘Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI-94)’, Amsterdam, The Netherlands, pp. 709–713.

Munroe, S., Luck, M. & d’Inverno, M. (2003*a*), Towards a motivation-based approach for evaluating goals, *in* ‘Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems’, ACM Press, Melbourne, Australia, pp. 1074–1075.

Munroe, S., Luck, M. & d’Inverno, M. (2003*b*), Towards motivation-based decisions for worth goals, *in* J. M. V. Marik & Pechoucek, eds, ‘Multi-Agent Systems and Applications III: Proceedings of the Third International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS 2003)’, Vol. 2691 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Berlin, Germany,, Prague, Czech Republic, pp. 17–28.

Myers, K. L. (1997), User Guide for the Procedural Reasoning System, Technical report, Menlo Park, CA.

Nau, D. S., Cao, Y., Lotem, A. & Mũnoz-Avila, H. (1999), SHOP: Simple Hierarchical Ordered Planner, *in* ‘Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence’, Morgan Kaufmann Publishers Inc., pp. 968–975.

Nau, D. S., Yang, Q. & Hendler, J. (1990), Optimization of Multiple-Goal Plans with Limited Interaction, *in* ‘Proceedings of the Workshop on Innovative Approaches to Planning’, Scheduling and Control, San Diego, CA, pp. 160–165.

Nebel, B. (1989), A knowledge level analysis of belief revision, *in* ‘Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning’, Morgan Kaufmann Publishers Inc., pp. 301–311.

Newell, A. (1994), *Unified theories of cognition*, Harvard University Press.

Norman, T. J. & Long, D. (1995), Goal creation in motivated agents, *in* ‘Intelligent Agents: Theories, Architectures, and Languages (LNAI Volume 890)’, Springer-Verlag: Heidelberg, Germany, pp. 277–290.

Obermarck, R. (1982), 'Distributed deadlock detection algorithm', *ACM Transactions on Database Systems (TODS)* **7**(2), 187–208.

Padgham, L. & Lambrix, P. (2000), Agent Capabilities: Extending BDI Theory, *in* 'Proceedings of Seventeenth National Conference on Artificial Intelligence - AAAI 2000', Austin, Texas, USA, pp. 68–73.

Paolucci, M., Shehory, O., Sycara, K. P., Kalp, D. & Pannu, A. (2000), A Planning Component for RETSINA Agents, *in* 'Proceedings of the Sixth International Workshop on Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL)', Springer-Verlag, pp. 147–161.

Pednault, E. P. D. (1989), ADL: Exploring the middle ground between STRIPS and the situation calculus, *in* R. J. Brachman, H. J. Levesque & R. Reiter, eds, 'KR'89: Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning', Kaufmann, San Mateo, CA, pp. 324–332.

Pokahr, A., Braubach, L. & Lamersdorf, W. (2003), JADEX: Implementing a BDI-Infrastructure for JADE Agents, *in* 'EXP - In Search of Innovation (Special Issue on JADE)', Vol. 3, Telecom Italia Lab, Turin, Italy, pp. 76–85.

Pollack, M. E. (1991), 'Overloading intentions for efficient practical reasoning', *Noûs* **25**(4), 513–536.

Rao, A. S. (1996), AgentSpeak(L): BDI agents speak out in a logical computable language, *in* 'Agents Breaking Away, Seventh European Workshop on Modelling Autonomous Agents in Multi-Agent World, MAAMAW '96', Vol. 1038 of *Lecture Notes in Computer Science*, Springer, Eindhoven, The Netherlands, pp. 42–55.

Rao, A. S. & Georgeff, M. P. (1991), Modelling rational agents within a BDI-Architecture, *in* R. Fikes & E. Sandewall, eds, 'Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning, KR '91', Cambridge, MA, pp. 473–484.

Rao, A. S. & Georgeff, M. P. (1992), An abstract architecture for rational agents, *in* C. Rich, W. Startout & B. Nebel, eds, 'Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning, KR '92', Morgan Kaufmann Publishers, San Mateo, CA, pp. 439–449.

Rao, A. S. & Georgeff, M. P. (1995), BDI-Agents from theory to practice, *in* 'Proceedings of the International Conference on Multi-Agent Systems (ICMAS-95)', MIT Press, San Francisco, USA, pp. 312–319.

Rao, A. S., Georgeff, M. P. & Kinny, D. (1996), A Methodology and Modelling Technique for Systems of BDI Agents, *in* W. V. de Velde & J. W. Perram, eds, 'Agents Breaking Away, Seventh European Workshop on Modelling Autonomous Agents in Multi-Agent World, MAAMAW '96', Vol. 1038 of *Lecture Notes in Computer Science*, Springer, Eindhoven, The Netherlands, pp. 56–71.

Rosenschein, S. J. (1985), 'Formal theories of knowledge in AI and Robotics', *New Generation Computing* **3**(4), 345–357.

Rosenschein, S. J. & Kaelbling, L. P. (1986), The Synthesis of Digital Machines with Provable Epistemic Properties, *in* 'Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning about Knowledge', Monterey, CA, pp. 83–98.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W. (1991), *Object-oriented Modeling and Design*, Prentice Hall.

Russell, S. & Norvig, P. (1995), *Artificial Intelligence: A Modern Approach*, Pearson Education.

Sacerdoti, E. (1977), *A structure for plans and behaviour*, American Elsevier.

Sathi, A. & Fox, M. S. (1989), 'Constraint directed negotiation of resource allocation', *Distributed Artificial Intelligence* **II**, 163–195.

Shoham, Y. (1991), AGENT0: A simple agent language and its interpreter, *in* 'Ninth National Conference on Artificial Intelligence (AAAI-91)', Anaheim, CA, pp. 704–709.

Shoham, Y. (1993), 'Agent-oriented programming', *Artificial Intelligence* **60**(1), 51–92.

Shoham, Y. & Tennenholtz, M. (1994), 'On social laws for artificial agent societies: off-line design', *Artificial Intelligence* **72**(1-2), 231–252.

Silberschatz, A., Galvin, P. B. & Gagne, G. (2001), *Operating System Concepts*, John Wiley & Sons, Inc.

Smith, D., Frank, J. & Jónsson, A. (2000), 'Bridging the gap between planning and scheduling', *Knowledge Engineering Review* **15**(1), 61–94.

Steeb, R., Cammarata, S., Hayes-Roth, F. A., Thorndyke, P. W. & Wesson, R. B. (1988), Distributed intelligence for air fleet control: Architectures for distributed air traffic control, *in* A. H. Bond & L. Gasser, eds, 'Readings in Distributed Artificial Intelligence', Kaufmann, San Mateo, CA, pp. 90–101.

Sugie, H., Inagaki, Y., Ono, S., Aisu, H. & Unemi, T. (1995), Placing objects with multiple mobile robots - mutual help using intention inference, *in* 'Proceedings of IEEE International Conference on Robotics and Automation', Vol. 2, Nagoya, Aichi, Japan, pp. 2181–2186.

Sycara, K. (1988), Resolving goal conflicts via negotiation, *in* 'Proceedings of the Seventh National Conference on Artificial Intelligence', Vol. 10, pp. 245–250.

Tambe, M. (1997), 'Towards flexible teamwork', *Journal of Artificial Intelligence Research* **7**, 83–124.

Tate, A. (1977), Generating project networks, *in* 'Proceedings of the Fifth International Joint Conference on Artificial Intelligence', pp. 888–893.

Tessier, C., Chaudron, L. & Müller, H.-J., eds (2000), *Conflicting Agents: Conflict Management in Multi-Agent Systems*, Kluwer Academic Publishers.

Thangarajah, J. & Padgham, L. (2004), An empirical evaluation of reasoning about resource conflicts in intelligent agents, *in* 'Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS 2004)', New York, USA, pp. 1298–1299.

Thangarajah, J., Padgham, L. & Harland, J. (2002), Representation and reasoning for goals in BDI agents, *in* 'Proceedings of the Twenty-Fifth Australasian Conference on Computer science', Australian Computer Society, Inc., Melbourne, Australia, pp. 259–265.

Thangarajah, J., Padgham, L. & Winikoff, M. (2003*a*), Detecting & exploiting positive goal interaction in intelligent agents, *in* 'Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems', ACM Press, Melbourne, Australia, pp. 401–408.

Thangarajah, J., Padgham, L. & Winikoff, M. (2003*b*), Detecting and avoiding interference between goals in intelligent agents, *in* 'Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003)', Acapulco, Mexico, pp. 721–726.

Thangarajah, J., Winikoff, M., Padgham, L. & Fischer, K. (2002), Avoiding resource conflicts in intelligent agents, *in* 'Proceedings of the Fifteenth European Conference on Artificial Intelligence 2002 (ECAI 2002)', Lyon, France, pp. 18–22.

Tidhar, G., Rao, A., Ljungberg, M., Kinny, D. & Sonenberg, E. (1992), Skills and capabilities in real-time team formation, Technical Report 27, Australian Artificial Intelligence Institute, Melbourne, Australia.

Tyrrell, T. (1994), 'An evaluation of Maes's bottom-up mechanism for behavior selection', *Adaptive Behavior* **2**(4), 307–348.

van Linder, B., van der Hoek, W. & Meyer, J.-J. (1995), Formalising motivational attitudes of agents: On preferences, goals and commitments, *in* M. Wooldridge, J. Müller & M. Tambe, eds, 'Intelligent Agents II: Agent Theories, Architectures, and Languages, IJCAI'95 Workshop (ATAL)', Springer, LNAI 1037, pp. 17–32.

van Riemsdijk, B., van der Hoek, W. & Meyer, J.-J. C. (2003), Agent programming in Dribble: from beliefs to goals using plans, *in* 'Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS 2003)', Melbourne, Australia, pp. 393–400.

Warren, D. (1976), Warplan: A system for generating plans, Memo 76, Department of Computational Logic, University of Edinburgh.

Weld, D. S. (1994), 'An introduction to least commitment planning', *AI Magazine* **15**(4), 27–61.

Weld, D. S. (1999), 'Recent advances in AI planning', *AI Magazine* **20**(2), 93–123.

Wilkins, D. (1988), *Practical planning: Extending the classical AI planning paradigm*, Morgan Kaufmann, CA.

Winikoff, M., Padgham, L., Harland, J. & Thangarajah, J. (2002), Declarative & procedural goals in intelligent agent systems, *in* 'Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002)', Toulouse, France, pp. 470–481.

Winslett, M. (1990), *Updating logical databases*, Cambridge University Press.

Wobcke, W. R. (2000), On the correctness of PRS agent programs., *in* Jennings, N.R. and Lesperance, ed., 'Intelligent Agents VI.', Springer-Verlag, Berlin, pp. 42–56.

Wooldridge, M. (2002), *An Introduction to MultiAgent Systems*, John Wiley and Sons, Chichester, England.

Wooldridge, M. & Jennings, N. R. (1994), Agent theories, architectures, and languages: A survey, *in* M. Wooldridge & N. R. Jennings, eds, 'Proceedings of the ECAI-Workshop on Agent Theories, Architectures and Languages', Amsterdam, The Netherlands, pp. 1–32.

Wooldridge, M. & Jennings, N. R. (1995), Intelligent agent : Theory and practice, *in* 'The Knowledge Engineering Review', Vol. 10(2), pp. 115–152.

Young, R. M., Pollack, M. E. & Moore, J. D. (1994), Decomposition and causality in partial-order planning, *in* 'Artificial Intelligence Planning Systems', pp. 188–194.