ANYTIME DELIBERATION FOR COMPUTER GAME AGENTS

by

NICHOLAS ANDREW HAWES

A thesis submitted to
The University of Birmingham
for the degree of
Doctor of Philosophy

School of Computer Science
The University of Birmingham
Birmingham, B15 2TT

November 2003

Abstract

This thesis presents an approach to generating intelligent behaviour for agents in computer game-like worlds. Designing and implementing such agents is a difficult task because they are required to act in real-time and respond immediately to unpredictable changes in their environment. Such requirements have traditionally caused problems for AI techniques.

To enable agents to generate intelligent behaviour in real-time, complex worlds, research has been carried out into two areas of agent construction. The first of these areas is the method used by the agent to plan future behaviour. To allow an agent to make efficient use of its processing time, a planner is presented that behaves as an anytime algorithm. This anytime planner is a hierarchical task network planner which allows a planning agent to interrupt its planning process at any time and trade-off planning time against plan quality.

The second area of agent construction that has been researched is the design of agent architectures. This has resulted in an agent architecture with the functionality to support an anytime planner in a dynamic, complex world. A proof-of-concept implementation of this design is presented which plays *Unreal Tournament* and displays behaviour that varies intelligently as it is placed under pressure.

Acknowledgements

First I must express my sincere gratitude to my supervisor, Aaron Sloman. Without his encouragement and knowledge, and the intellectual freedom he allowed me, I would not have been able to tackle this research.

Perhaps the greatest influence over the day-to-day direction of my research was provided by my first external supervisor, Ian Wright. Although I lost his official supervision just as I was starting to make progress, he continued to motivate and guide me with an equal mix of inspiration and fear. For his unfaltering input and friendship I am truly grateful.

None of this would have been possible without the support and funding of Sony Computer Entertainment Europe, and the work of Dave Ranyard. Also, many thanks must go to Manfred Kerber and Mark Ryan for their insightful comments as members of my thesis group.

I would also like to thank my family and friends who have supported me from the very beginning of my education until now, especially those that were there for me during the writing-up period. The list is too long for everyone to be named individually, but if I've spoken to you in the last four years, then you've helped me through this.

I don't think I would have even managed to survive the first year of my PhD without the always welcome distractions provided by my like-minded office mates. For the rowdy nights out and racquet sports, thanks to you all.

I would like to take this opportunity to thank my A-Level Physics teachers, Mr (John) Atkinson and Mr (Andy) Stoddart, who made me work when I really needed pushing.

Finally, I offer all of my love and thanks to Rosie for moving to Birmingham, putting up with me, and supporting me through everything.

Contents

List of Figures

List of Tables

Part I

Preamble

Chapter 1

Introduction

Computer games have come a long way in the last forty years. From humble beginnings such as *Spacewar*[1], *Space Invaders* and *Donkey Kong* [Poole, 2000, Chapter 2], they have grown to be the focus of a multi-million pound industry. To sustain such a large interest, game development companies are constantly looking for ways to advance the technology used in games. In recent years, the idea of pursuing academic research into Artificial Intelligence (AI) for computer games has gained popularity. This has resulted in a community of researchers looking to apply and develop AI techniques that are directly applicable to computer game worlds. Computer games are an ideal domain in which to pursue AI research for a number of reasons (these are set out in Section 1.2.1), but primarily because their typically character-based nature makes them a natural arena for the application of agent-based techniques. An agent has been defined as many things in the past, but for this thesis we will use the term to define anything that can perceive and act in an environment. By taking this approach we can view almost any character in a computer game as an agent.

The aim of this thesis is to explore what techniques make it possible to design and implement an agent specifically for computer game worlds. To constrain this aim further, the particular agent we are attempting design must display intelligent behaviour in order to enrich the player's experience of the game. As the term "intelligent behaviour" is not only vague, but variable depending on domain and agent, we will define it as any behaviour which is *goal-oriented* (i.e. acted out in the pursuit of a particular goal or goals), yet flexible. Such behaviour demonstrates intelligence because it shows that the agent has an awareness of the situations it desires to be in, and what changes it can affect to bring about such situations [Hawes, 2000]. Goal-oriented behaviour is one of the defining characteristics of an intelligent agent [Franklin and Graesser, 1997, Wooldridge and Jennings, 1995].

The remainder of this chapter starts with a brief description of what a computer game is, and then sets out some reasons why games are suitable for AI research. This is then followed by a game scenario to demonstrate the kinds of problems the thesis aims to solve. Finally, the main contributions made by this thesis are presented and the structure of the remaining thesis is outlined.

[1] *Spacewar* is one of the earliest examples of a computer game. It was implemented on a PDP-1 in 1962 [Poole, 2000].

1.1 What is a Computer Game?

For the remainder of this thesis, a computer game is defined as a virtual world with which one or more users (or players) interact in order to achieve a particular goal. A game world is usually populated with agents that either help, hinder or are neutral to the player or players. A player's interaction with the world occurs in real-time, so that when a player manipulates the world to change it, the effects of this change are immediately acted out (e.g. if the player moves to open an unlocked door then the door opens immediately). Although this description may not cover all existing computer games (e.g. turn-based strategy games), it is sufficient to cover the majority of cases that this thesis is concerned with.

From the above description we can emphasise two important points about the virtual worlds used in computer games. First, because a number of players can manipulate objects in the world, the world can be said to be *interactive*. We will take this to mean that, from the point of view of an agent in the world, the state of the world can change due the actions of other agents (not always itself). Second, because the world must respond immediately to the manipulations of the player, the world can be said to be *real-time*. As a consequence of this, an agent's response to an event must be acted out as swiftly as possible to maintain the illusion of a real-time world (this in turn maintains the believability of the game world).

A final point that can be made about computer game worlds is that because they are virtual simulations of existing or imaginary environments, they can be arbitrarily *complex* (see [Firby, 1989, Chapter 1] for an additional definition of this term). This means that any agent intended to be placed in a computer game world must be able to deal with arbitrary levels of complexity.

It is interesting to note that these common features of computer game worlds are also features of the real world. This immediately demonstrates how research carried out in the virtual world of computer games can be applicable in a wider context.

1.2 An Introduction to AI Research for Games

The general theme of this thesis will be the application of AI techniques to the field of computer games. Before the focus is narrowed to deal specifically with the problem of agent design and construction, it will be informative to further examine the reasons for carrying out AI research in the computer games domain. These reasons stem from the potential benefits to academic AI research and computer game development of investigating new AI techniques for games. Because these groups have different, and occasionally conflicting needs, they will be discussed separately.

1.2.1 General academic AI research

Computer games have a number of features which make them appealing as a domain in which to pursue AI research. Computer game worlds are very similar to the testbeds developed by many researchers [Atkin et al., 1999]. For example, herding games such as *Herdy Gerdy*[2] [Core, 2002] and *Sheep* [Mind's Eye, 2000] use domains similar to the one used by the Synthetic Characters group at MIT (e.g. [Isla et al., 2001]). Other examples include sports domains (e.g. *FIFA2002* [EA Sports, 2001] and the RoboCup [Kitano et al., 1997]), war games (*Command & Conquer: Generals* [Electronic Arts Pacific, 2003] and COASTER [Atkin et al., 1999]) and flight simulators (*Microsoft Combat Flight Simulator* [Microsoft, 2002] and Jones and Laird's automated pilots domain [Jones et al., 1998]). This similarity between game domains and existing AI testbeds means that by using games as test beds, researchers can tackle new problems in implemented simulations (i.e. the game worlds) without having to come to terms with completely new problem domains. This list of comparisons also demonstrates the wide variety of subject matter covered by computer games. Researchers looking to tackle a particular set of virtually realised physical conditions will not have to search for long before finding a game that provides them.

Unfortunately, although many games are ideal environments for AI research, very few actually provide access to the internal features that would allow researchers to contribute to what is already in a game. As a genre, First Person Shooters[3] (FPSs) are a notable exception to this trend. Due to the growth of online multiplayer games, a demand appeared for artificial opponents to practise against offline (in order to avoid getting summarily beaten when venturing online). To allow more opponents (or "bots" in game playing slang) to be developed to practise against, FPS developers allowed players to access APIs (Application Program Interfaces) that encapsulated sensing and acting procedures for the bots. The popularity of this feature has led to API access being available in many recent games (e.g. *Quake II* [id Software, 1997], and *Half-Life* [Valve, 1998]). Other games provide less powerful facilities for extending AI via scripting languages (e.g. *Baldur's Gate* [BioWare, 1998], and *Age of Empires* [Ensemble Studios, 1997]).

Using game worlds in preference to custom test beds has a further benefit: computer games already have a variety of useful features built into them. For example, physics and collision detection for solid objects, and vision and basic actions for agents. This allows researchers to immediately tackle the problems they are interested in (e.g. agent co-operation) without having to build their agents from the ground up, reinventing and reimplementing several "wheels" in the process.

A further benefit of pursuing AI research in the computer games domain is that the character-based nature of most games encourages researchers to tackle AI issues that relate to whole agents rather than disembodied systems (perhaps by situating their systems in an information processing architecture [Sloman, 2002]).

A less theoretical benefit of pursuing AI research in games is that it has the potential to raise the public profile of academic AI, by being highly visible in an industry that worldwide generates over

[2]In this thesis, computer games will be treated like any other published work and thus have an entry in the bibliography. Where a normal published work would traditionally have an author, a game will have a developer. The title, year, and publisher of a game will be presented in the same way as the corresponding features of any other published work. To distinguish games in the text of the thesis, their names will be printed in italics.

[3]A genre in which all action is viewed "through the eyes" of the agent being controlled, and in which action usually constitutes shooting other similarly equipped agents.

seventeen billion dollars worth of business a year [Poole, 2000, Chapter 1].

Because computer game worlds are simulations of arbitrary complexity, certain simulations will present problems to AI algorithms that approximate problems present in the real world. The advantage this provides is that solutions to problems in such game worlds will also be applicable to the same problems in the real world.

The final benefit of pursuing AI research in games is also a principal motivating factor of this thesis: AI techniques in most games must operate in real-time, dynamic environments. This means that any AI technique applied in a computer game domain must be able to work within these domain constraints. This will advance the current state-of-the-art in AI. For example, techniques controlling agents within games must be able to respond to stimuli before the game player senses a delay, and decision making processes must be aware that the state of the world can be changed at any time by other agents (including players).

1.2.2 Commercial game development

There are two principal reasons why commercial game developers would benefit from pursuing research into AI for games. The first, and possibly most desirable reason is that better game AI should result (if applied correctly) in better games[4]. This in turn should generate increased sales for a company, and hence higher revenue. Currently, most development companies are trying to produce games with more detailed graphics, but the industry is rapidly approaching a situation where increases in graphical effort are only providing diminishing returns on that effort. Because of this, other avenues must be explored to differentiate games in the market place. One of these avenues is AI.

The second reason for commercial interest in AI research into games is the ability to make game development faster and easier. This is particularly relevant for character AI. Currently a great deal of character AI is coded as extensive conditional structures. Programming these structures is time-consuming, as every possible situation has to be anticipated and an appropriate response has to be created (further discussion of reactive behaviour generation is presented in Section 3.2.2). AI techniques can improve upon this situation in a number of ways. For example, machine learning techniques (e.g. case-based reasoning [Aamodt and Plaza, 1994]) could be used to allow the agent to discover appropriate stimulus-response pairs itself, or generative planning (see Chapter 3) could be used to allow a character to generate its own behaviour from scratch using domain knowledge encoded into operators (although both of these examples present new problems and trade-offs of their own).

At this point it is important to highlight a conflict that is present between academic AI researchers and commercial game developers. For game developers, AI is seen as a tool for presenting an experience to a player. Methods for implementing character behaviour do not have to be domain independent or biologically plausible (two interesting academic AI concepts). They must be fast, computationally cheap and able to play a role within the designer's vision for the game. In contrast, AI researchers want to

[4] Although this is presented as a straightforward relationship, the interdependencies between various game components (e.g. game design, user interface and level design) mean that it is hard to witness an improvement in one game component without an improvement in a number of others.

investigate novel possibilities for things like agent behaviour generation (including methods for sensing and acting), and are interested in furthering knowledge. These almost orthogonal purposes (AI as a tool and AI for the sake of knowledge) lead the different fields in different directions. Developers look to the short term, favouring tried and tested techniques that come with guarantees. Such an approach will never significantly push back the boundaries of knowledge. AI researchers take a more "blue skies" approach, and free from commercial constraints, they look to the future and produce systems that are novel and interesting. Unfortunately, such systems will never find a place in games within the foreseeable future due to the aforementioned commercial constraints.

To overcome this conflict (i.e. to get more advanced AI into commercial games), compromises are needed from both parties. AI researchers who are pursuing game oriented research must be prepared to evaluate their work not only in terms of novel contributions to knowledge, but also in terms of how useful their work will be to game developers in the near future. For their part, games developers must appreciate the gains that will come with advanced AI, and look to the future when considering how to implement the AI for their next project. It would be pointless for one side to compromise more than the other because this would mean that they would fail to produce work that met their specific aims (useful game tools and contributions to knowledge), but with some compromise hopefully such conflicting aims can find a middle ground.

1.2.3 Agent-specific AI research for games

The preceding sections have discussed the motivations and benefits of researching AI for games in general. As this thesis will deal specifically with agent behaviour, it is necessary to address the motivations and benefits of pursuing this topic.

In general terms, the problem being addressed by the approaches investigated in this thesis is "what should an agent do next?". The motivation for this problem is that in the majority of situations (whether in games, or in other AI domains) the agent should be continually acting to achieve its goals. Goals can be present in an agent in a variety of forms. These can include long-term, ever-present goals which the agent must maintain (e.g. staying alive), regularly re-occurring goals based on the current state (e.g. finding food or reproducing), and sporadically occurring goals based on the agent's current behaviour (e.g. picking an apple, or opening a door). Such goals are not wholly independent of one another. Usually there is a hierarchical link between them, although this is not always explicit within a situation or agent implementation. Using the examples given above, the goal to survive will cause a goal to eat to be generated at regular intervals, which could in turn lead to a specific goal of picking an apple.

Determining what an agent should do next concerns all of the steps from the top to the bottom of this goal hierarchy. It involves translating abstractly stated, high-level goals (whether explicitly generated by an agent or implicitly coded into its behaviour) into specific instances of goals to achieve the desired behaviour. One of the advantages of investigating this process in computer game environments is that the goals of an agent are usually well defined (e.g. protect this tower, or find this item).

An additional benefit of games-related agent research is that all new techniques must effectively

Figure 1.1: A screenshot from a Capture The Flag game in *Unreal Tournament*.

deal with (or at least be mindful of) the issues presented by the games domain. These include the fact that all agents are required to respond in real-time to stimulus and to deal with unpredictably changing environments. In turn, this will allow more academic AI approaches to be applicable to "real world" situations.

1.3 An Example Game Scenario

To motivate the rest of the thesis, and to situate the research within a tangible framework, this section presents a scenario which introduces a specific game domain and some possible agent behaviours. Specific problems that this scenario presents are discussed in Sections 2.3 and 2.4.

The scenario is called *Capture The Flag* (CTF), and is based around a type of game found in many multiplayer computer games. It is a game played between two or more opposing teams, where each team is comprised of one or more players. The aim of the game is to either score a set number of points, or to have the most points when the game ends (after a fixed amount of time). Each team has a base, on which a flag rests. To score a point, a player must take the opposing team's flag from their base and then get it back to their own base. This action will only score a point if the team's own flag is at their base when the player returns with the opposition's flag. A screenshot from a CTF game in *Unreal Tournament* [Epic Mega Games, 1999] can be seen in Figure 1.1. The screenshot shows an agent carrying the opposition's flag whilst being attacked by opposition agents[5].

Agents in the CTF scenario can only perform a small number of actions. These are limited to running and jumping, picking up and putting down objects, and using weapons (this is typical for the FPS genre). In the CTF scenario, typical behaviours for agents include guarding their team's flag (to prevent the opposition from stealing it), attempting to steal the opposition's flag (in order to score a point), trying to prevent the opposition from scoring with a captured flag (attempting to "kill" the flag carrier) and assisting a team mate that has captured the opposition's flag (perhaps by attacking the opposition agents chasing it).

How a typical CTF scenario works can be illustrated with an example. Yigal (an agent in the CTF scenario) is trying to come up with a plan to score a point. He spends some time thinking about possible alternative courses of action, but then notices his team mate Milind defending their base from two opponent agents. Realising that he has a chance to capture their flag whilst Milind keeps the opposition busy, Yigal stops thinking about a possible plan and just acts on what he has come up with so far. This leads him to run directly to the opposition's base and grab their flag. He faces no resistance because the opposition are still attacking Milind. Bearing in mind the pressure Milind must be under, Yigal rushes back to their base, avoiding the opposition as much as possible, and manages to score a point. This example demonstrates a few of the previously mentioned behaviours in context, and hopefully provides a little insight into the types of problems that this thesis will have to address if it is to succeed in creating an agent that can behave intelligently in such a domain.

The scenario of Capture The Flag in an action-oriented game was chosen for this research for a number of reasons. First, it is quite a well known game style in the computer games community. Second, the majority of games that have AI accessible agent interfaces are FPSs, and CTF game types are almost universal in this genre of game. The need to implement the research in an actual game was an important factor in choosing the scenario domain. It would be unconvincing to claim that a developed technique would be effective in a computer game world, when it had only been tested in a purpose-built world. Therefore, a domain from a computer game had to be used. A final reason to choose CTF as a game domain for the scenario is that it presents a level of complexity above that of the typical "kill one another until the time runs out" scenarios found in many other games.

Even with these advantages, there are reasons why the FPS-based CTF domain is flawed as a testing ground for AI techniques. First, although it is more complex (and hence challenging for the applied techniques) than a number of other types of game, it falls short of the level of complexity represented by many existing AI testbeds. As such, it provides a less than adequate test of the agent's problem solving abilities. A second, and possibly more important reason is that success in CTF is usually determined by physical skills such as aiming ability and reaction times, rather than cognitive abilities (cf. [Cavazza, 2000]). On account of this, a purely reactive agent (e.g. [Agre and Chapman, 1987]) would generally perform better at most of the important aspects of the game, rather than the hybrid agent that is

[5] All of the agents in this screen shot are controlled by implementations of the designs developed in this thesis.

subsequently introduced by this thesis. The previously noted simplicity of the CTF game augments the advantage a reactive agent would have over a deliberative or hybrid agent.

1.4 Summary of Main Contributions

The work in the rest of this thesis presents two principal contributions to knowledge. The first of these is an original method for agent plan generation that is flexible with respect to processing time and result quality. In addition to this, the method is ideally suited to integration in an information processing architecture for an intelligent agent. These qualities are provided by designing the plan generation method as an *agent-oriented* (see Section 3.2.1) *anytime algorithm* (see Chapter 4). Although the first main contribution is this approach to plan generation, further contributions to knowledge are made during the design of the approach and the analysis of past attempts to achieve similar aims.

The second principal contribution made by this thesis is a novel agent architecture designed to support this flexible plan generation method and successfully cope with the domain features of computer game worlds (i.e. real-time interactivity and complexity). The design of the agent is developed through an iterative methodology that aims to explore *design space* (see Section 2.1). This example of such a design methodology in action constitutes a secondary contribution of the thesis.

As the motivations, requirements and designs for the main contributions are expounded, further secondary contributions are made. These are not presented here due to their more technical nature, but are summarised along with the main contributions in Section 10.1.

1.5 Overview of Thesis

The body of the thesis is split into four parts. The current chapter and following chapter constitute Part I. This provides an introduction to the problems being tackled by the subsequent parts. Part II deals almost exclusively with the issue of agent-oriented planning in computer game worlds. Part III then expands the focus of the thesis to investigate how an agent can be designed for computer game worlds. Finally, Part IV offers some overall conclusions on the whole thesis. In more detail, the contents of the subsequent chapters are as follows;

- Part I
  - Chapter 2 expands on the general introduction given in this chapter to provide a more detailed problem statement for the rest of the thesis.
- Part II
  - Chapter 3 reviews some existing literature on planning. This review pays particular attention to approaches that may be suitable for application in computer game worlds.

  - Chapter 4 introduces the concept of anytime planning (i.e. planning as an anytime algorithm), then goes on to review existing literature on the subject. This is followed by an attempt to highlight features common to all anytime planning approaches.
  - Chapter 5 details the design of the Anytime Universal Method Composition Planner (A-UMCP), an anytime planner that supports true anytime behaviour and is developed specifically to be integrated into an agent architecture.
  - Chapter 6 presents experimental analysis of A-UMCP and its component technologies.

- Part III

  - Chapter 7 reviews literature concerned with the design of intelligent agents. The review is primarily concerned with hybrid agents and existing approaches to developing agents for computer game worlds.
  - Chapter 8 charts the iterative development of a design for a computer game agent. Changes made in each iteration are guided by experimental evidence from a proof-of-concept implementation of the design.
  - Chapter 9 analyses the performance of a proof-of-concept implementation of the agent designed in the previous chapter. The analysis focuses on the advantages the agent gains by planning in an anytime manner.

- Part IV

  - Chapter 10 offers conclusions on the thesis in the form of a summary of contributions and critical analysis. Some possible future directions for the research are also presented.

Chapter 2

Detailed Problem Statement

In this chapter, the problems faced by behaviour generation methods in computer game worlds are elaborated. The reason for doing this is to narrow the current view of the aims of this thesis, until we are left with a manageable problem. Before this is done, the methodological approach of the thesis is presented.

2.1 Methodology

Research in Artificial Intelligence can be pursued in a variety of ways. Relevant discussions include [Beaudoin, 1994, Section 1.3] and [Sloman, 1993b]. This thesis will follow in the footsteps of previous related work (e.g. [Beaudoin, 1994] and [Wright, 1997]), and follow a design-based methodology. A design-based methodology is an approach to explaining how systems (biological, non-biological, existing, hypothetical etc.) work, based on how they are designed (cf. Dennett's design stance [Dennett, 1978]). When designing agents to exhibit particular behaviours, such a methodology promotes a particular style of agent design. The requirements for the design should first be ascertained (e.g. the behaviours the agent must exhibit), then designs can be drawn up to satisfy these requirements. These designs should then be tested and revised in order to better fit the original requirements.

The design-based construction of systems can be undertaken in many different ways. The following five steps expand upon the brief description given above and provide some useful guidelines to follow when working within a design-based methodology.

1. Determine the requirements of the system you are trying to design. These requirements serve a similar purpose to a specification in a software engineering project. At this point, the functionality required by the design should be specified. This determines the *niche* or role that the proposed design is intended to occupy (for discussions of the relationship between design space and niche space see [Sloman, 1995] and [Sloman, 1998]).

2. Produce designs which satisfy the requirements set out in step one. When applying this approach to agent design, the proposed designs will be detailed layouts of agent architectures. Further details on agent architectures are presented in Section 2.5.

3. Implement the design or designs produced by step two. The implementation of the design may utilise any available medium (e.g. hardware or software). The implementation is intended to allow the designer to discover unseen flaws or potential benefits present in the design. An implementation will also lead to a better understanding of how the potentially complex parts of the design interact. The implementation process may also provide a deeper understanding of the requirements of the design, as (possibly abstract) design concepts are made concrete.

4. Evaluate how well the implementation represents the design, and to what degree this design meets the requirements specified in step one. This evaluation can done by experimental or analytical means.

5. Examine the *design space* surrounding the implemented design. This step will involve studying how changes to the design would provide additional functionality that may or may not allow it to better meet the specified requirements. It is only by comparing the design to other similar designs (i.e. those that occupy neighbouring design space) that we can develop an understanding of particular design features (e.g. when comparing the trade-offs between two designs that differ on one feature) [Sloman, 1995].

Although the steps are discretised here to allow clearer explanation, the steps can overlap or be performed in parallel. When this is the case, later steps can feed back into earlier steps, modifying the overall direction of the research.

These steps are roughly mimicked by the structure of this thesis. The remainder of this chapter represents step one. As such it presents an effort to determine the requirements of the system we are trying to design and understand (an agent for a real-time computer game). These requirements will be as detailed as possible for this stage of a design, but will be refined later as possible designs are introduced (thus begins the first of the step overlaps). Once the requirements for the agent are specified, possible designs (for both whole agents, and component parts) are discussed. This is done in terms of previous work that proposes designs that meet aspects of the requirements. Once these have been taken into account, an agent design is proposed, and an example implementation of the design is introduced. Both the design and the implementation are then subject to analysis and revision as the above steps are iterated through a number of times.

2.1.1 Simulation vs. Entertainment

Relevant to the methodological approach of the whole thesis is the trade-off between simulation and entertainment. Simulation is the approach typically taken in AI research, with systems being developed that aim to simulate or replicate some form of existing (or proposed) intelligence in a valid manner (where the notion of validity is relevant to the research domain, e.g. whether the method could have evolved biologically). This can be generalised by stating that simulation-based AI is more concerned with the knowledge and techniques used to develop the system (and make it "valid") than the end product (e.g. an example implementation of an agent). In contrast to this, when developing agents for entertainment purposes (e.g.

for games) the sole goal is to produce something that provides a particular experience to the user. When developing entertainment agents, any technique that provides the desired behaviour can be used, whether it is scientifically valid or not. In the entertainment domain, a valid approach to developing an agent is any approach that creates an entertaining agent. The intellectual tension between approaches that aim to simulate and approaches that aim to entertain is broadly comparable to the relationship described in Section 1.2 between commercial and academic computer game AI development.

A good example of difference between the simulation and entertainment approaches is the use of the Gibsonian notion of affordances[1] [Gibson, 1986]. Any valid simulation (i.e. AI research) approach to developing a system that recognised and took advantage of environmental affordances would have to deal with a wide variety of visual processing and subsequent reasoning (e.g. [Sloman, 1994]), and such a system would ultimately have to be able to form part of an encompassing cognitive architecture. In contrast to this, the notion of an affordance is the basis of the agent behaviour in the commercial game *The Sims* [Maxis, 2000]. Rather than attempt to simulate the recognition and use of affordances by giving the agents suitable mechanisms, affordances are hard-coded into the objects in the environment. This means that by inspecting an object an agent in *The Sims* immediately knows how to use the object and what it can use it for.

As an instance of AI research, this thesis will take the simulation approach to agent development. Whilst not necessarily aiming to produce mechanisms that reflect certain processes in humans (and other animals), this thesis aims to produce something that at least could be considered to simulate certain aspects of human behaviour (most notably plan construction under pressure). Because this research is positioned in an entertainment domain, the results of the research will be evaluated in a suitable context (a game), although the question about whether the end products of the research provide an entertaining game experience will not be addressed in detail. This is because determining what is "entertaining" and why, is a topic that falls beyond the limits of this thesis.

2.2 Expanded Scenario

To enable us to specify the requirements of the system we are designing, this section will introduce a detailed example of a typical scenario. The scenario is based on a similar situation to the brief scenario in Section 1.3, but will feature some additional behaviours and will be presented in a greater detail than was earlier necessary. This additional granularity will give us further insight into the functionality required from the inner workings of an agent in a computer game world.

A rough overview of the scenario is as follows; the agent (Yigal again) is planning to score a point, but his team's flag gets captured so he stops planning, he then starts planning how to retrieve the flag, and ends up interrupting this planning process as well in order to attempt to recover the flag before the opposition score a point. This scenario has been chosen because it encompasses not only the basic abilities required from the agent, but because it also has specific instances when the agent has to deal

[1] In brief, Gibson's theory of affordances proposes that organisms sense objects in the world in terms of what the objects offer to the organism.

with the dynamism of the environment.

The scenario starts with Yigal in a place of relative safety in the game. He is embodied within the context of the game, rather than being an external controller for an avatar[2]. His first act is to appraise the current state of the game. This appraisal includes inspecting both the knowledge that has been stored previously regarding positions of objects in the game world (e.g. ammunition or medical packs) and the up-to-date knowledge about the state of the game (e.g. the score, and whether the team flags are safe). From this appraisal, a goal is proposed for Yigal to pursue. How this goal is proposed is not important yet, just the fact that a mechanism is present that proposes goals. As he has no current goal, Yigal accepts the proposed goal as the aim of his subsequent course of action. Next, Yigal has to determine what actions are necessary to achieve this goal. To do this he examines the current state of the world and determines what actions he can currently perform and how they will bring him closer to his goal. He then considers what he could do in the world after he had performed that previous action. This continues until Yigal has produced a sequence of actions that will achieve the goal. Whilst constructing a plan of action, Yigal must constantly be aware of his surroundings in case something changes in the environment that affects the goal that he his planning for. Such a change occurs during the time that Yigal is creating a plan to score a point; the opposition steal his team's flag. It is possible that other changes may also occur within his environment, but he has some way of distinguishing between critical changes (e.g. his team's flag being stolen) and non-critical changes (e.g. his team-mate picking up a weapon). Once a critical environmental change has been noticed, Yigal has the ability to promptly redirect his limited resources towards solving the new problem (in this case, how to get the flag back). This redirection of resources results in a new plan construction phase. Part of Yigal's monitoring during this phase involves being aware of how long it usually takes to construct and execute a plan to achieve his current goal. He also has some idea of the amount of time it usually takes for the opposition to score a point with the flag. In this instance, Yigal becomes aware that the time required for him to retrieve the flag (i.e. finish the plan then execute it) is about to become longer than the predicted time until the opposition score. This causes the plan construction process to be interrupted, and Yigal receives the partially finished plan. At this point Yigal assumes that the plan contains enough information to guide him to recapture the flag (or he has a mechanism to fill in the gaps), and proceeds to execute it. During the execution, Yigal has to contend with the fact that the plan was compiled using static assumptions about the environment. As such, he may have to account for changes in the environment whilst still attempting to complete steps from the plan (e.g. the flag may have moved).

[2]"Embodiment" in the agent domain usually means that an agent is in possession of a *physical* body. In this case, although the agent is simulated within a world and not physically embodied, it is embodied with respect to the simulated world it is in. This means that the agent is restricted to a single physical object in the world, only receives the sense data that the object would receive in the physical world, and can only act upon the world using the object.

2.3 Planning Problems In Detail

2.3.1 Problems from the scenario

From the previous scenario, we can identify processes that specifically relate to the way in which Yigal formulates the behaviour which will enable him to achieve his goals. This processing will be known as "planning" for the remainder of the thesis.

1. Yigal builds plans based on current knowledge of the world he is in, and his current internal state.
2. Alternative ways of achieving a single goal are considered.
3. The plan Yigal creates is executable in the world he planned in, even though some changes to the world may have occurred.
4. The process of planning can be interrupted when Yigal decides that it is more important to plan for something else.
5. If Yigal feels it is important to achieve a goal in a hurry, then the planning process can be interrupted to retrieve a partially constructed plan to be immediately executed.

2.3.2 Problems in general

From the abilities that Yigal demonstrated in the scenario, we can identify some general problems that planning will face in domains such as the one Yigal is in.

Problems 1 and 2 from the above list, formulating a plan from an initial set of facts and considering alternative courses of action, are problems that are solved by all planning approaches. Because of this, they will not be discussed in detail. Planning provides a way of ordering actions that are executable by an agent (known as *primitives*) in order to achieve a goal. This process of constructing a plan from a set of primitives can be done in a variety of ways. An introduction to planning is given in Section 3.1. Problem 3 has been tackled in numerous of ways by a variety of researchers. For example, predictable world changes have been dealt with through the use of conditional plans (e.g. [Pryor and Collins, 1993]) or by allowing other mechanisms to fill in the necessary details as the plan is executed (e.g. [Malcolm, 1997] and [Aylett et al., 1995]). Problems 4 and 5 are behaviours that have traditionally not been covered by the planning field. They are important to this thesis because the planning that is taking place is embodied within a complex agent, which is in turn embedded within a world that is changing regularly. The changeable nature of computer game worlds heavily influences the majority of problems that are tackled by the agent. The next section describes how the qualities of typical game worlds (see Section 1.1) will affect any planning process situated within such worlds.

2.3.3 Planning in dynamic worlds

A crucial aspect in both of the scenarios presented previously (Sections 1.3 and 2.2), one that influenced the majority of Yigal's behaviour, was that a number of actions were going on in parallel. Some were visible to Yigal (e.g. one of his teammates was defending their base), others were not (e.g. a member of the opposition team sneaking up to steal Yigal's team's flag). Because of these manifold, possibly overlapping or conflicting threads of behaviour present throughout every session of the game Yigal is playing, it is difficult for him to predict exactly what is going to happen in the future. This problem generalises from the given scenario to the majority of computer games (and the real world). The presence of agents with motivations and states that are not accessible to the agent (e.g. computer controlled adversaries, or human players) means that there is every chance the agent's world will change in potentially unforeseeable ways, and at potentially unforeseeable times.

This observation correlates with one of the key characteristics of computer game worlds that were identified in Section 1.1. The world is *interactive* (in real-time). This refers to the fact that the world can be changed by the agents within it. These interactions are not limited to the agent doing the planning (Yigal in the scenarios); any agent can affect a change in the world given some prerequisite abilities. In addition to this, the world is *dynamic*. In this case, the word is used to refer to an extreme case of interactivity. Where the term "interactive" refers to the fact that the world can change, "dynamic" imputes that the world *will* change, and will do so frequently. This leads to the world being inherently unpredictable as it is impossible for the agent to accurately (or otherwise) predict where other agents will be and what they will do. This definition of the word "dynamic" has been used previously, e.g. [Firby, 1989, Chapter 1].

When Yigal constructs plans for the future, the scenario stated that he did this based on the current state of the world by considering how the world changes through the application of possible actions. When we examine the possibility of an agent constructing plans in this manner in an interactive, dynamic world, we can highlight a number of flaws with this approach. Many of these flaws are discussed in [Pollack and Horty, 1999] along with further analysis.

- **Invalid initial assumptions:** Because the world may have changed without the planning agent being aware of the changes, it is possible that the initial state used to plan from may be incorrect. Because the world may no longer satisfy the preconditions for actions that were assumed during the planning process, this may lead to a plan being flawed upon execution.

- **World change during planning:** It is possible that the world can change during the planning process. The longer the planning takes, the greater the chance of this happening. This again leads to a situation where the resulting plan may not be executable.

- **World change during execution:** If the planner completes without the world changing, the final plan is still based on static assumptions about the environment it is to be executed in. This means that changes to the environment may again render the plan inexecutable.

- **Global goal change:** Related to the previous flaws, it is possible that in a dynamic world, the goals of the agent that initiated the planning process may change significantly during planning or execution. This may result in the plan or planning process being viewed with a different priority by the agent. This behaviour presupposes a number of things about the planning agent. It is beyond the scope of an isolated planning process, and will be further discussed in Section 2.4.

These problems present barriers to designing a planner to generate agent behaviour in a computer game world.

2.4 Agent Problems In Detail

2.4.1 Problems from the scenario

We can now identify specific problems that Yigal had to deal with that are beyond the scope of a hypothetical planner in its architecture. The question of how and where to draw the distinction between the roles of the planner and the rest of the agent is critical when designing an agent architecture. The factors that determine this distinction are related to the dynamism of both the goals the agent is trying to achieve, and the world in which the agent is trying to achieve them.

The following list identifies some of the more complex problems solved by Yigal (or the implementation of Yigal) in the scenario. It ignores some low-level assumptions that, whilst possibly relevant to the overall design of an agent, are beyond the scope of problems to be discussed in this thesis (e.g. sensing, motor control, object differentiation etc.).

1. Yigal must be able to know the current state of the game (scores, time limit etc.).

2. Yigal needs to be able to store information about locations of objects in the world.

3. Yigal can execute a precompiled series of actions.

4. Yigal can extract salient information from the complex world around him.

5. Yigal can decide which goals are important to achieve in the future.

6. Yigal can prevent other goals of great importance (e.g. staying alive, stopping the other team scoring) from overriding his current goal (e.g. scoring a point).

7. Yigal is able to monitor his environment during the process of planning.

8. Yigal is aware of the typical lengths of time that his actions, and the actions of others, take.

9. Yigal can stop planning for future behaviour and just act on what has been formulated so far[3].

[3]This is a problem for the agent, rather than the planner, because the agent must have the mechanisms to control the planner.

2.4.2 Problems in general

A number of general agent design problems can be drawn from the previously presented specific examples.

- **Sensing:** The agent must be able to sense things in its environment and interpret these sense data appropriately. It must also be able to receive feedback on its internal processing. If the environment is particularly complex then it is important that the agent is able to direct its sensory apparatus towards the objects in its environment that are significant to its behaviour.

- **Knowledge management:** The agent must be able to store knowledge about its environment in an appropriate form, be able to recall it when necessary, and delete it when it is no longer appropriate. This raises a problem similar to the one presented by sensing; what information should an agent store. Problems related to this include what format should knowledge be stored in, how it should be indexed for easy retrieval and when is it no longer useful to store particular facts.

- **Goal management:** An agent must be able to determine which goal it should pursue in order to maximise some kind of future utility measure (this could be anything from a point score in a game, to length of time spent alive). This kind of decision may be non-trivial, especially if a number of competing goals are available for the agent to pursue.

- **Plan execution:** The agent must be able to take a plan produced with reference to some static assumptions about its environment (See Section 2.3.3) and execute it in an environment where those assumptions may be incorrect.

- **Resource redirection:** As all computer game agents will have limited processing resources, the agent needs to be able to direct its resources to solve the current problems it faces and not waste resources on irrelevant tasks. This problem is related to some of the previously mentioned problems (selective sensing and knowledge storage).

These problems present barriers to designing an agent that will exhibit intelligent (and believable) behaviour in a computer game world. Previous attempts to overcome these problems will be presented in Chapter 7.

2.5 The Architectural Approach

Many of the above problems cannot be tackled in isolation. For example, how an agent senses its environment has a direct effect on how it stores data internally, and how an agent builds its plans has a knock-on effect on how they can be executed. If an agent for a computer game is to be developed, it must be able to overcome such problems. To do this, individual components of the agent (e.g. its planner) must be designed to form part of an agent architecture.

In basic terms, an architecture defines what components are used to construct an agent, and how these components are connected. An introduction to some general classes of architecture can be found in [Sloman and Logan, 2000]. When viewed in detail, architectures can provide information about what data are passed and shared between subsystems in an agent and the precise routes of information flow through an agent. When viewed at a lower granularity, the architectural structure defines constraints on the variety of possible behaviours that could be generated by the components of which the agent is composed. The behaviour of information within the architecture can be interpreted in terms of concepts typically used to refer to mental states and processes. For example, in [Wright et al., 1996] grief is explored in terms of perturbances within an architecture. Such an interpretation is not always valid though, and the applicability of a particular interpretation may depend on the design of the agent architecture.

To produce an agent that displays intelligent behaviour in an environment that provides numerous obstacles to generating such behaviour (e.g. those features of a typical game world, as described in Sections 1.1 and 2.3.3), it is not enough for the components of an agent to be designed with these environmental features in mind. It is critical that the components are designed to be an effective part of an encompassing architecture. This may require components to provide additional feedback about internal processes, allow access to control mechanisms, or present data in a particular way. If this is not done, then the architecture may have to exert (i.e. waste) additional resources to integrate parts of its architecture, or have its behaviour unnecessarily constrained by components of its architecture not effectively working together towards the current goal of the agent. Examples of possible badly integrated architectures include an architecture that has a set of reactions which may generate behaviours that conflict with an agent's attempt to follow a plan to achieve a goal, or an architecture that stores all knowledge in a form that is unreadable by its components and hence requires an intermediate translation stage before any information is used.

2.6 Detailed Problem Statement

In light of the previous discussion, the problems to be tackled within this thesis can now be stated in slightly greater detail. The final direction of the thesis is still as described previously; to design an agent that can demonstrate intelligent behaviour in a complex, dynamic, real-time computer game world. In order to achieve this, two major problems must be tackled.

The first of these problems is to develop a planning algorithm suitable for an agent in a computer game world. The design of the planner must be informed by the knowledge that the agent is constantly under time pressure when constructing plans, and that the resulting plans are to be used to guide the agent's behaviour in an ever-changing world (rather than being static solutions to more abstract problems). The design of the planner must also acknowledge that the agent will require as much control as is feasible over the performance of the planner. The agent should not be subservient to the planner (waiting patiently while a plan is produced), but the planner should be fully integrated into the agent's architecture.

This leads to the second major problem faced by this thesis. An agent architecture must be developed that can deal with the demands of a computer game world. Again the design must be informed by knowledge of the common features of game worlds, and the problems these cause to the typical processing tasks required of the agent. In addition to this, the developed agent architecture must provide the necessary features to support and control the planner produced in response to the problems of planning in a computer game world.

The remainder of the thesis explores these problems. This exploration involves expanding upon both the problems that have been presented and their relationship to the thesis as a whole, and proposing designs for architecture components and whole architectures which satisfy the requirements identified during the specification of the problems. During this process, further sub-problems are identified that are not general enough to be presented at this early stage.

Part II

Planning For Computer Game Worlds

Chapter 3

Review of Planning Literature

This chapter reviews previous work in the field of planning. As it is impossible to do justice to the complete history of planning in the limited space available, attention is paid primarily to planning work that is related to the objectives of this thesis, at the expense of other issues. The review starts with the basics of planning, then moves on to ways in which some of the problems presented by planning have been tackled. Reactive planning is reviewed as a method of generating fast, robust behaviour and continual planning, a sub-domain of planning that has goals similar to those of this thesis, is introduced. Finally, other techniques that might be useful for developing a planner to be used by an agent in a world that is complex and dynamic are presented.

3.1 A Brief History of Planning

Planning in its most basic form is the construction of a series of actions to form a plan that will take the plan executor from an initial state to a desired goal state. States in a planning process are usually described by a conjunction of literals. The grandparent of all modern planning systems was the General Problem Solver (GPS) [Newell and Simon, 1969][1]. GPS worked on a general notation that could represent various problems (not just planning problems, e.g. theorem proving). It started with an initial state and tried to reduce the difference between this and a goal state through the application of logic-based rules. GPS was followed by the arrival of the currently most well known planning system, STRIPS [Fikes and Nilsson, 1971]. STRIPS introduced a form of action representation that is still used today, the planning *operator*. An operator is a representation of an action *primitive*, and forms the basic building block of many varied planning systems.

An operator has three key features; a precondition list, an add list, and a delete list. An example planning operator to move a block from the top of one pile of blocks onto another pile can be seen in Figure 3.1[2]. The preconditions of an operator express a logical formula which must evaluate to true in

[1] GPS was originally developed in the 1950s.

[2] This is an operator from the Blocks World, a domain introduced in the late 60s and used to demonstrate many different problem solving approaches. It was originally presented in [Winograd, 1972].

```
Move (a, b, c):
     Precs: On(a, b)∧Clear(a)∧Clear(c)
       Add: On(a, c)∧Clear(b)
    Delete: On(a, b)∧Clear(c)
```

Figure 3.1: A STRIPS style operator to move block a from block b onto block c.

order for the operator to be applicable. This represents the necessary features that must be present in a state before the action can be performed in that state (e.g. a block must be clear before it can be moved). The add list represents the facts that are added to the state after the operator is applied. Adding facts to a state aims to capture the positive effects of performing an action (e.g. after moving, the block is now on another block). The delete list is similar to the add list, but it represents the facts that are removed from the state after the application of the operator. This represents the negative effects (i.e. things that are no longer true) of performing an action (e.g. the block onto which the moved block is placed is no longer clear).

Many planning systems (starting from GPS) build plans through means-ends analysis. At a high level this involves inspecting what needs to be done (i.e. what goals need to be achieved), and what is the gap between the current state and the goals. Operators are then applied to make the gap between the goal state and the initial state smaller. The idea behind this method is demonstrated in the following excerpt from [Newell and Simon, 1972].

> I want to take my son to nursery school. What's the difference between what I have and what I want? One of distance. What changes distance? My automobile. My automobile won't work. What is needed to make it work? A new battery[3].

Planning using means-ends analysis with individual executable actions has been termed *classical* or *traditional* planning. The above example represents *linear* planning, with one step being considering directly after the preceding one. Planning can also be done in a *non-linear* manner (e.g. UCPOP [Penberthy and Weld, 1992]), in which goals are considered without committing to a fixed ordering of actions. Non-linear plans can be either linearised to produce plans similar to to those produced by linear planners or they can be left in a non-linear state to retain generality or to be executed by multiple entities.

Traditional planning is an effective method for solving small, static problems, but it has flaws that prevent wider applicability. The most critical of these is that in a variety of situations (e.g. those requiring a complete and optimal plan), planning has been shown to be intractable [Chapman, 1987]. The consequence of this result is that traditional planning methods lack the scalability needed to be applicable to the complex domains that characterise modern planning problems (e.g. the logistics domain in [Andrews et al., 1995]). The intractability of planning does not prevent planning from being a viable

[3]Whilst this example uses backward chaining (i.e. what action achieves a particular goal), means-ends analysis can also chain forward (i.e. what action can I take in the current state) or in both directions at once.

problem solving method, though it does mean that thought should be given to the types of problems being tackled with planning and the way these problems are represented.

As was discussed previously (see Section 2.3.3), traditional planning also has a number of flaws that are evident when attempting to apply its methods to real-world situations. The previously mentioned flaws will not be reiterated here, but some of the other unmentioned ones will be presented (including some from [Pollack and Horty, 1999]). These are;

- Actions in a plan have completely certain outcomes. In real life and virtual worlds actions have uncertain outcomes (i.e. they may not have the desired effect).
- Actions in a plan have instantaneous effects in the environment. In real and virtual worlds the effects of an action may take time to occur, if the action is effective at all.
- Planning goals are either satisfied or unsatisfied, no partial satisfaction is considered (cf. the use of "soft constraints" in [Logan and Alechina, 1998]). Again this is at odds with the real world, in which notions of partial satisfaction exist.
- Traditional planning does not deal with conditionals or loops. Such structures naturally form the building blocks of certain real-world behaviours (like hammering a nail until it is fully embedded in a plank of wood).

These problems (along with the previously identified ones) must be overcome by either the planner or the planning agent if a planning agent is to be implemented in a computer game domain.

The complexity of planning problems has been dealt with in various ways. One of the most widely used methods of managing complexity is *abstraction*. The term abstraction in planning usually refers to a method of removing detail from the planning problem. Abstraction has been applied to planning in at least two distinct ways; *precondition removal planning*, and *hierarchical task network planning*.

Precondition removal planning (e.g. [Sacerdoti, 1974] and [Yang et al., 1991]) involves abstracting away complexity by ignoring preconditions from STRIPS-style operators, then gradually adding preconditions back in to refine the solution until all preconditions are considered and satisfied. By only considering a subset of the preconditions at any time, a less complex problem is presented to the planner. This allows a skeletal plan to be constructed quickly which can then be used to direct the subsequent planning. In [Knoblock, 1994] a method is presented for automatically generating precondition-removal abstraction hierarchies from state-based problem descriptions.

The second approach to removing complexity from planning problems through abstraction is hierarchical task network (HTN) planning [Tate, 1977]. HTN planning abstracts away detail in a different manner from the previous approach. Instead of constructing plans from primitive building blocks (as in classical planning), HTN planning removes details from planning problems by combining many actions into one less detailed (more abstract) action termed a *task*. Task networks are collections of tasks that have certain constraints associated with them, and can be considered the hierarchical task network planning equivalent of a plan. Examples of task network constraints include ordering constraints over

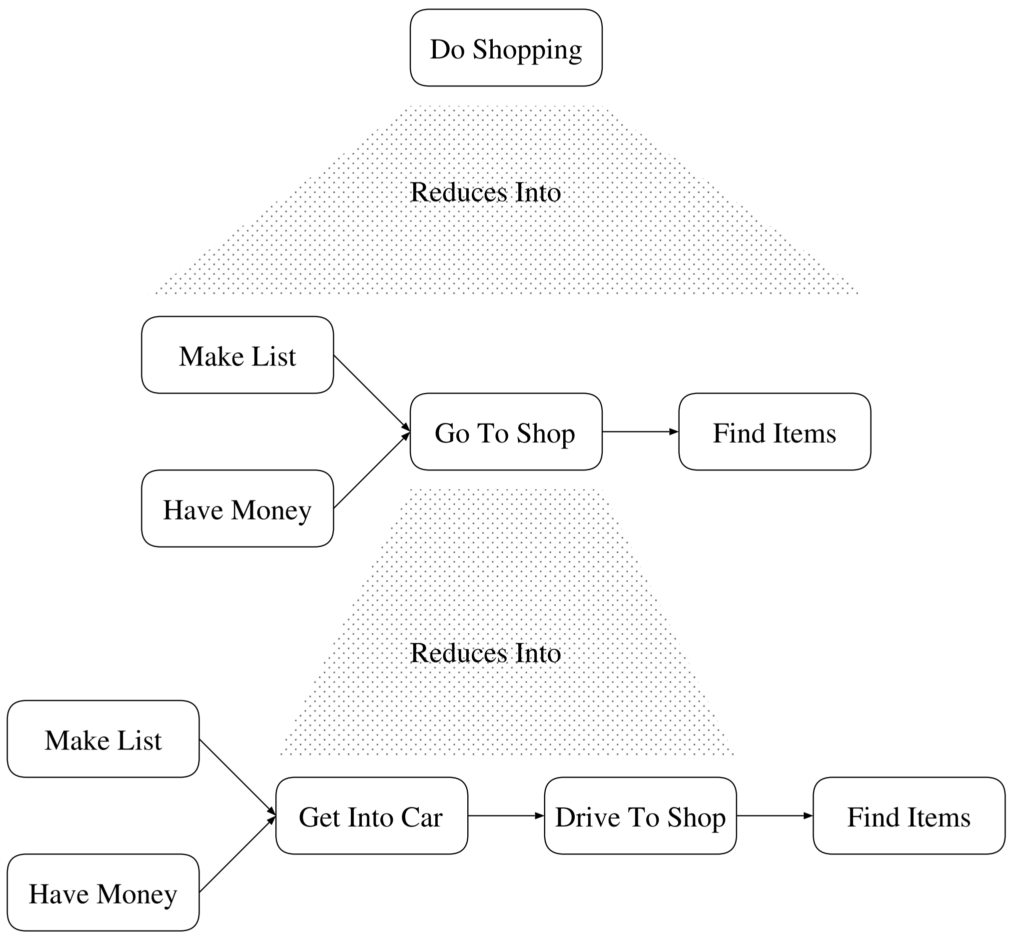

Figure 3.2: An example of reducing a Hierarchical Task Network.

tasks and variable constraints within individual tasks. Tasks can be generated automatically (e.g. learning from completed plans to form macro operators (MACROPS) was done in a later version of STRIPS [Fikes et al., 1972]) or through a designer examining the planning domain (e.g. as with the tasks for UM Translog [Andrews et al., 1995]). HTN planning constructs plans by reducing (i.e. breaking down) abstract tasks in a task network into more detailed tasks until no further reductions can be made. At this point, the task network should be completely composed of the basic actions performable in the planning domain (the primitives). The rule or procedure used to reduce an abstract task into a set of less abstract tasks is called a *method*. The idea behind this process is demonstrated in Figure 3.2. The figure shows the abstract task `Do Shopping` being reduced into less abstract tasks that make up part of the shopping task. It then shows a further reduction of `Go To Shop` into tasks that are less abstract again (i.e. have less detail removed). HTN planning involves a great deal of domain specific knowledge (i.e. which tasks reduce into which other tasks) and this, combined with the reduction in search space through the use of abstract tasks, helps simplify solving complex planning problems. HTN planning is discussed in more detail in Chapter 5.

3.2 Planning Approaches Suited to Game Worlds

After the previous presentation of some background on planning, we can now start to investigate existing work that may be suitable for use as a component of an architecture for an intelligent agent in a computer game.

3.2.1 Fast planners

Although the process of planning has traditionally been slow (as planning algorithms haven't adequately handled the inherent complexity and intractability of planning), recent advances in the state-of-the-art, as showcased in the bi-annual planning competition (e.g. [McDermott, 2000]), have brought much greater speed (i.e. better methods of coping with complexity) to general purpose planners. Probably the most revolutionary planning development of recent years has been GraphPlan [Blum and Furst, 1997]. The techniques used in GraphPlan have been built upon to produce a new generation of fast graph-based planners (e.g. IPP [Koehler, 1998] and Blackbox [Kautz and Selman, 1999]).

Most of these modern planners are powerful enough to solve the planning problems in the Capture The Flag scenario in a few seconds or less, although more complex domains would still require longer. If speed was the sole issue when planning in a computer game world, then using any of the aforementioned planners would satisfy part of the aim of this thesis. Unfortunately, as we have seen already, plan generation is only a small part of the whole package of behaviour generation for a computer game agent.

General purpose planners are specifically designed for solving planning problems in isolation. Although a blackbox approach to problem solving may be successful in certain domains, an agent in a dynamic world must have more control over its planner than just starting it and waiting for a solution. Ideally a planner for use in an agent should be a fully integrated part of its architecture. Integration into an architecture may require additional planner feedback and control interfaces, and perhaps even changes to the underlying algorithms that take advantage of the encapsulating architecture (e.g. data from execution traces could be used to guide search away from known problem areas, or certain operators could contain gaps to be filled in at execution time). To put this idea succinctly, a move away from *solution-oriented* planning (is the solution optimal? is the planner complete?), and towards *agent-oriented* planning (is behaviour being generated? is the behaviour achieving the goal?) is required.

The class of solution-oriented planners encompass almost all existing deliberative planners. Their development is typically focused on either solving a specific problem or class of problems. Solution-oriented planners usually aim to find a solution in the smallest possible number of steps, or the shortest amount of time, whilst also looking for the shortest possible plan or the plan that uses the least of a particular resource. Solution-oriented planners are planners that are implemented purely to solve disembodied problems in the best possible way, without taking into account factors that do not directly effect the planning process. In contrast, agent-oriented planners are more focused on their role in a larger system. The design of agent-oriented planners should encompass notions of how they can be controlled by the system they are embedded within (e.g. an agent architecture) and how their output is going to be used (e.g. what is the plan used as input for). Agent-oriented planners should still aim to be fast and produce optimal plans, but should also offer flexibility to controlling processes. For example, they could allow the controlling process to ignore optimality in order to increase planning speed or vice versa, or to plan for a particular conclusion at the expense of all others. Using a solution-oriented planner in an agent architecture may result in extra work for the agent designer (e.g. the ACT language had to be developed to mediate between the planner and executor in the Cypress agent [Wilkins et al., 1995]), or a necessary absence of agent actions during the planning process due to a lack of control (e.g. the agent from the Oz project must wait whilst the planner runs for a set amount of time [Blythe and Reilly, 1993]). The only examples of generally applicable agent-oriented planners that exist are all reactive planners (see Section 3.2.2). Reactive planners are usually chosen when a flexible approach to planning is required because they offer a fast solution that can be easily tailored to specific problems.

A final issue with planners implemented purely for speed is that however fast a planner is, it will have to work within the CPU and memory restrictions of the platform it is implemented on. On modern PCs running a single planning process this is not an overly restrictive constraint. But in a game environment, running multiple planning processes in parallel (for multiple agents) within a game that is already tying up the majority of resources on a machine, it may be hard to satisfy the resource requirements of almost any planner. Because of this, a planning approach that is more flexible and controllable with respect to resource usage is required.

3.2.2 Reactive approaches

To overcome the weaknesses of planning deliberatively in interactive and dynamic worlds, the majority of researchers tackling such domains have turned to reactive or behaviour-based solutions for generating future actions. Such techniques (e.g. the work presented in [Agre and Chapman, 1987] and

[Brooks, 1986]) have typically been applied to the creation of complete agents based on these paradigms, rather than planning components for agents. We will narrow the focus of this review to deal with those techniques that are intended to form part of an agent, rather than comprise the whole thing. A discussion of techniques used to build complete reactive agents is presented in Section 7.1.

The following example from [Firby, 1989, Section 1.1] demonstrates the purpose to which reactive planning is usually put.

> In my apartment, the TV set is in the bedroom so I am usually there too. Getting a glass of water means walking out of the bedroom, through the living room, and into the kitchen. Along the way I will usually have to walk around furniture and packages placed randomly by my wife and myself. I may also have to avoid the cat which sometimes jumps out unexpectedly. Finally, in the kitchen there will be a glass to find somewhere in the cupboard and dishes to be moved out of the way in the sink before I can pour a glass of water. [...] This simple task cannot possibly be planned to more detail than "go to the kitchen", "get a glass from the cupboard", "fill it with water at the sink".

Rather than plan for the three behaviours mentioned at the end of the quote, reactive planning aims to provide behaviour solutions for these goals that are based directly on the observable state of the world (e.g. the positions of cats and packages).

The general concept behind all reactive planners is that the predictive complexities of generative planning can be avoided by having a set of predetermined actions that will achieve a particular goal given a particular state. The sensory inputs of the agent are combined with a goal to produce a mapping to the correct set of actions for the current situation (possibly including a change of internal state in the reactive planner). Usually this is guided by some over-arching control structure to tell the planner which goal to tackle next. This generalisation is not taken from a particular source, but is summarised from the literature on approaches to reactive planning.

In classical planning terms, reactive planning is equivalent to a mapping from the current and goal states to a plan that achieves the goal state. This mapping will change as the current state changes during plan execution (e.g. the movement of Firby's cat requires a change of path through his living room). This technique is epitomised by Schopper's Universal Plans [Schoppers, 1987]. The Universal Plan system works in the following way. Before execution time, a non-linear planner is used to generate plans, which are then compiled into a decision tree. At runtime the decision tree is checked against the current state of the world and the correct plan is retrieved. This runtime checking is the only online (i.e. during the agent's active "life") processing that the agent has to do, and requires considerably less CPU time than full generative planning.

Two implementations of reactive planning that are similar in philosophy to Universal Plans are the Reactive Action Package (RAP) system [Firby, 1987], and the Procedural Reasoning System (PRS) [Georgeff and Ingrand, 1989]. Where Universal Plans employ one structure (the decision tree) to encode the variety of ways of achieving a goal, both the RAP system and PRS break the knowledge into smaller specialised units that can be manipulated by an interpreting process. Each of these specialised units has particular invocation conditions, and a goal or goals which they will subsequently achieve. The units within the RAP system (named RAPs themselves) can contain two types of knowledge; execution instructions to achieve the desired goal, or a subgoal to be achieved by another RAP. These types of knowledge can also be combined in one package. Plans are generated in the RAP system by recursively interpreting RAPs (i.e. either executing instructions or interpreting the sub-goal RAPs) in a manner that is similar to traversing the decision tree of a Universal Plan.

PRS allows more sophisticated reasoning than the previous methods by having a special type of Knowledge Area (KA) (the name used in PRS for the units of knowledge it manipulates) called a metalevel Knowledge Areas. Metalevel KAs contain arbitrarily complex information about how to manipulate other KAs and the current state of the system. Such an ability allows PRS to overcome the ballistic nature of Universal Plans and the RAP system and change the way it generates behaviour at runtime, at the expense of additional processing costs (as such meta processing starts to act like planning). Using the previous water-fetching example we can highlight the difference between Universal Plans and both PRS and the RAP system. Once a Universal Plan for fetching water is being followed, only events anticipated within the plan can be acted upon (e.g. the position of obstacles), but if something unanticipated within the water-fetching scenario occurs (e.g. the cat won't stop bothering the agent because it needs feeding), then the agent acting on a Universal Plan won't be able to deal with it. In contrast, if the agent has a RAP or PRS Knowledge Area to deal with feeding the cat, then it can switch to achieving this before proceeding on to fetching some water.

Reactive planning appears to be an adequate method of behaviour generation for an agent in a real-time, dynamic world. Indeed, the majority of non-player characters in current computer games are implemented in a similar manner. By precompiling responses to situations, reactive planning agents can avoid potentially time consuming generative planning. Also, by basing action decisions on only the current state, the agent is able to quickly react to changes in the environment.

To get these advantages, reactive planners must make certain trade-offs. For example, the saving in execution time comes at the expense of the time spent compiling the reactive behaviours and the storage space taken up by the behaviours. In the Universal Plans system, this offline compilation is automated by the use of a non-linear planner, but many other systems (including PRS and the RAP system) default to the reactive behaviours (RAPs, KAs etc.) being composed by the designer of the agent. Designing reactive behaviours is a difficult task, as careless development can lead to situations where behaviours can get stuck in feedback cycles and other unhelpful interactions. Other flaws with designed behaviours include not coping with enough situations, and making planning rules too coarse or fine grained (too much or too little detail to differentiate between situations). A final issue is the reliability of reactive planning when dealing with goals. The water fetching example can be used to highlight this. Reactive planning is employed to handle short-term goals that have potentially dynamic components, but some other architectural component is providing the longer term goals (e.g. "go to the kitchen"). Reactive planning has difficulty manipulating longer term goals, particularly when the path to them is complex (the above example contains only quite simple goals). In PRS and the RAP system, the designer has to overcome this by anticipating possibly complex goal interactions when writing the behaviours. In the

Universal Plan system, the non-linear planner is used to anticipate goal interactions.

Although it appears that reactive planning has side-stepped the complexity issues associated with generative planning, it also has its own problems with scalability and complexity. Ginsberg argues in [Ginsberg, 1989] that for an agent to have a plan for every possible course of action requires an exponential growth in the complexity of processing required to match the plan to the current state. Producing detailed behaviour requires a huge look up table of possible actions, thus introducing a new source of complexity (finding the correct action to perform). This is not to say that reactive planning is not useful, but it is important to identify the kinds of problems that it should be applied to. For example, Ginsberg offers a bad application in the form of a reactive chess playing program, and suggests that highly physical problems (e.g. manipulating blocks) are better suited to being solved reactively. This is because it is the physical aspects of the world that usually change the fastest, and hence require reactivity. This is true, although there are many examples of non-physical aspects of the world that change at a rapid pace, e.g. the thoughts in someone's head.

Although the examples offered by Ginsberg are shallow, they accurately capture the distinction between reactive and deliberative planning. Problems that have an unpredictable or highly dynamic component (e.g. physically manipulating blocks) are best tackled reactively. Problems that have complex goal interactions (e.g. chess) or require monitoring over the length of their solution (e.g. producing a plan that minimises the use of a particular resource) are best tackled deliberatively. Any task that features a mix of these problem types will not be adequately tackled by either deliberative or reactive processing. Instead, it will require a mix of these approaches (e.g. a robot that plays chess, planning its moves deliberatively then moving the pieces reactively).

3.2.3 Continual planning

A subset of planning directly concerned with planning in dynamic environments is continual planning. Continual planning embraces the philosophy that planning is "an ongoing, dynamic process in which planning and execution are interleaved" [desJardins et al., 1999]. It aims to introduce solutions to the problems that are caused when the traditional planning assumptions are abandoned. It typically involves constructing a planning architecture featuring interacting components that support a central planner, rather than developing a particular planning algorithm[4]. Continual planning represents a step towards the previously introduced idea of agent-oriented planning because it shows an awareness of problems beyond the scope of pure plan generation.

An attempt to devise a continual planner for a domain directly analogous to computer games is presented in [Atkin and Cohen, 1998]. The domain is the computer simulation of military actions. The authors are working on a problem that consists of two teams of agents aiming to capture all of the opposing team's flags (a larger scale version of the scenario presented in Section 1.3). Their work makes two interesting contributions. The first contribution questions the way in which planning algorithms represent the world. Planning methods typically describe the world in terms of discrete states. This does

[4]This is similar to placing the planning algorithm into an agent architecture.

not accurately describe the way events occur in the real world. In reality, performing an action takes a certain length of time. This length of time may overlap other events or have exploitable fluctuations. The authors suggest that a lot of information is lost about the world when we impose the artificial boundaries of states. Although it is easy to see the truth in this, it is a lot harder to actually go beyond a discrete state representation. One way to extend the standard approach to planning to present a more realistic view of the world is to associate actions with time values, allowing the planner to reason about the possible effects of overlapping actions and other possible relations (see [Long and Fox, 2003] for the current state-of-the-art of temporal representations for planning).

The second contribution made by Atkin and Cohen is the use of plans as general schemas for action. Instead of using a generative planner to create a new plan every time one is required, a planner is used offline to construct a number of plans to cover a variety of situations. These plans are then generalised as plan schemas and stored for reuse as KAs[5] in a modified version of the PRS reactive planning system (reviewed in Section 3.2.2). This allows the agent to reuse the plan in any situation which can be unified with the preconditions of the associated KA. This method of plan reuse takes advantage of the fact that there is typically only a small number of solutions to any problem. This is different from normal PRS because this modified version is storing whole plans (i.e. combinations of PRS KAs) for use at runtime, rather than individual actions (single KAs).

Although the use of plan schemas circumvents the run-time complexity inherent in generative planning (the complexity is moved to rule writing and matching), it also means that the flexibility of generating a plan from scratch at run time is lost. At best this may mean that the agent will miss unexpected opportunities that are unique to the current situation. This could happen if the plan schema for the situation did not allow for such opportunities, as may happen if the opportunities are sufficiently rare to not be included in a general case plan. At worst this may mean that the agent has no plan to execute because the current situation and goal do not match any of its stored plans. This could happen if an unpredicted occurrence alters the world in such a way that none of the precompiled plans are applicable. Situations such as these will be more likely to occur as the domains being planned in get more and more complex. An increase in domain complexity will also mean that more schemas are required to adequately cover all possible combinations of goals and situations. If all the possible plans were generated for a complex domain (perhaps using a deliberative planner before run time) the storage space required for them, and the effort required to select the appropriate one, would be very large. This is identical to the scaling problems inherent with reactive planning (see Section 3.2.2). Problems aside, the use of plan schemas represents a useful, complexity reducing technique that could be used in conjunction with many planning systems, especially if they were to be used in a domain with known limits (cf. proof planning to reduce complexity when searching for mathematical proofs [Bundy, 1988] or the similarly purposed MACROPS extension to STRIPS [Fikes et al., 1972]).

Another implementation of a continual planning system in a military domain (air combat) is the Continuous Planning and Execution Framework (CPEF) [Myers, 1999]. CPEF is a continual planner which is composed of a number of interacting modules which provide facilities for interleaving planning

[5] As mentioned previously, a Knowledge Areas (KA) is a PRS term for a reactive rule.

and execution and real-time conservative repair of the current plan. Its central component is a plan manager. This sends commands to the other modules (such as the planner and the plan repair module) and also monitors the execution of the plan in the world. One interesting feature of CPEF is the way it keeps alert to possible problems with the current plan. CPEF allows the automated creation of event-response rules called *monitors*. Monitors are generated to provide a specific response given the occurrence of a particular event in the environment (cf. special and general purpose monitors described in [Sloman, 1978, Chapter 6]). There are three main classes of monitor in CPEF, these are:

- **Failure**: Provide an appropriate response to a specific failure in the plan.
- **Knowledge**: Test for the availability of required information about the world.
- **Assumption**: Detect whether an assumption made during the planning process has been violated.

Although these monitors are constructed purely for planning and plan repair purposes, it is clear to see that they share common ground with the problems identified in Section 2.4 which must be solved by a game agent design.

In [Pollack and Horty, 1999] the authors lay out the necessary features of a (continual) planning system that is intended to operate in the real world. They list;

- **Plan generation**: A method for initially developing plans.
- **Commitment management**: A way of reasoning about what level of commitment to attribute to a plan and its associated goals.
- **Environment monitoring**: Keeping alert to possible opportunities and threats.
- **Alternative assessment**: A way of assessing conflicting opportunities.
- **Plan elaboration**: An ability to take partial plans and then elaborate them when more information is available.
- **Metalevel control**: Decide how much time to devote to particular planning problems.
- **Coordination with other agents**: How to plan when other agents (that may have threatening, helpful or neutral goals) may act in the world.

These features echo the problems that were extracted from the scenario in Section 2.4.

3.2.4 The Autodrive project

The Autodrive project [Wood, 1993] was a project to design agents that could independently drive a car and navigate in highly dynamic environments (the dynamism comes from moving agents and unexpectedly blocked roads). Although not strictly concerned with planning, the work on the project makes some interesting contributions related to this thesis, particularly in the analysis of planning in a dynamic and real-time environment. As this analysis has a wider focus than just planning, the concepts introduced here will be useful for both the planning and agent design contributions of this thesis.

Wood identifies a dichotomy of types of information in an agent's environment. The first side of this dichotomy is information that remains constant in the long-term, or only changes very slowly. Examples of this type of information in the domain of driving are positions of roads (i.e. maps), the permanent capabilities of the agent, and how certain things can be achieved in the environment (e.g. positions can be changed from driving from A to B). The other side of this dichotomy of information in an agent's environment is information that changes in the short-term[6]. Examples of this are the positions of mobile agents, the state of objects in the environment, and the current motivation of the agent. Wood states that it is inefficient to maintain one representation that includes both these of types of data. Instead, a technique of *dynamic world modelling* is proposed [Wood, 1993, Section 4.5]. This technique dictates that a permanent representation should be used for the long term information. This can be used for planning, and for reasoning about the long term data. When the agent needs to make judgements based on the current state of the environment (or anything that is not long-term information) a representation should be generated specifically for this purpose. This representation should contain all the short-term data necessary for the decision to be made. Once the decision has been made, the representation can be "forgotten" by the agent.

Dynamic world modelling has the advantage of cutting down the processing overhead of continually updating a representation of the entire world with a large number of variables, especially when the variables are likely to change before they can be used. A deficiency of this approach is that if reasoning needs to be done quickly, generating the necessary representation may take so long that the time for reasoning has passed. It may also be desirable to integrate dynamic data with long term data in some domains.

The idea of having two different types of information in an environment is also used as the platform for Wood's second important contribution to planning in a dynamic world; goal-subgoal instability. When an agent creates its plans using a hierarchy of abstractions based on the current world state (e.g. hierarchical task network planning, see Section 3.1), the higher levels of abstraction should be based on the long-term and hence static features of the world. Between these upper levels of the hierarchy there exists a relationship that the author terms *goal-subgoal stability*. This means that the links present between one level of the hierarchy and another (representing the refinements of abstract tasks) are unlikely to change (they are stable). Because of this stable relationship the agent can have confidence that this part of the plan will be valid in the future. The opposite of this is true nearer the bottom of the hierarchy. These levels are characterised by *goal-subgoal instability*; refinements of abstract tasks, and sequences of necessary primitive actions are likely to change regularly. For example, an agent may be safe to plan a route between two cities based on the road layout because information regarding the position of roads is very stable (although interesting problems arise when this information is incorrect and the agent is part-way through its journey). On the other hand, if the agent plans its journey by using the positions of

[6]This is just one type of division of information based on types of change. For example, information could be divided based on whether changes are discrete or continuous, or add or remove complexity.

other cars as reference points, it is extremely unlikely that this plan will stay valid as the data it is based on will become almost immediately out of date.

To exploit these relationships, the agent (or agent's designer) should determine at which level of abstraction the world starts to become dynamic (named the *interface level* in [Wilkins et al., 1995]), and only represent the planning problem above this level. To plan below this level represents a waste of processing because the more specific dynamic information can change at any time, making the plan inapplicable. Also, by only representing knowledge to this level, the search space, and hence the planning problem's complexity, is greatly reduced. To execute the resulting abstract plan, the agent can use reactive actions that encode the behaviour represented by the abstract tasks at the interface level. This allows the agent to fill-in or elaborate the dynamic implementational details of behaviour (e.g. positions of objects, method of locomotion etc.) at run-time. By taking advantage of goal-subgoal stability in this way, dynamic reactive behaviour can be situated within a static goal-oriented framework. This is one possible approach to situating a fairly traditional planning approach in the real world (or a simulated version of it).

The ideas from the Autodrive project, particularly those related to the concept of goal-subgoal instability, capture the intuition behind how it appears that humans form plans. We create general solutions at a relatively high level of abstraction with stable concepts and not much detail. Some aspects of the high level structure can be filled in prior to execution, but which bits largely depend on the exact domain. As the plan is executed the precise details are filled in when knowledge becomes available, with some of the final details being amended through feedback during actions (e.g. the precise angle to turn the wheel at when manoeuvring around a parked car). The idea of goal-subgoal instability subsumes other ideas about instantiating pre-built plan structures at run time (e.g. MACROPS in STRIPS [Fikes et al., 1972] and the plan schemas presented in [Atkin and Cohen, 1998]). All of these approaches aim to avoid the complexity involved in building plans at low levels of details by grouping actions in larger combinations without specifying information that cannot be reliably inferred when the pre-built structures are constructed.

These concepts, particularly the run-time instantiation of plans and the abstract representation of actions, will be revisited in Chapter 8 when designs are considered for placing a flexible planner in an agent architecture.

3.3 Summary

In this chapter we have seen some selected approaches that have previously been taken towards behaviour planning. Both classical planning and fast, stand-alone planners have been judged as inadequate to the task of generating behaviour for an agent in a dynamic, real-time environment. Some widely-used reactive planning techniques were reviewed, but were also found to have weaknesses tied to scalability and applicability. The idea of continual planning was presented as a step away from pure, solution-oriented planning and towards agent-oriented planning (a term introduced in Section 3.2.1 to refer to the type of planning component that would form a useful and practical component of an agent architecture).

Finally, some ideas on knowledge representation for planning in a dynamic domain were reviewed. These ideas were based primarily on the concept of a division between the stable and unstable facts in an agent's world.

Chapter 4

Introduction to Anytime Planning

An anytime algorithm is a type of algorithm that is ideally suited for flexible, real-time performance. This chapter starts by introducing anytime algorithms in general terms, then discusses the advantages and disadvantages of implementing a planning algorithm as an anytime algorithm. After this, existing approaches to planning in an anytime manner are reviewed, and some general observations about these approaches are made.

4.1 What is an Anytime Algorithm?

In the broadest terms, an anytime algorithm is an algorithm that can be interrupted at any time during its execution and will always return a valid solution to the problem it is solving. The term is due to Dean and Boddy who originally suggested the concept of anytime algorithms in their two papers [Dean and Boddy, 1988] and [Boddy and Dean, 1989]. Their ideas were further developed by others including [Zilberstein and Russell, 1993], [Mouaddib and Zilberstein, 1995] and [Zilberstein, 1996].

Although an algorithm that can be stopped at any time is potentially useful, a guarantee on its performance is necessary to ensure its applicability. This guarantee is that the quality of the solution returned by the anytime algorithm will never decrease as processing time increases. This makes the result quality of the anytime algorithm monotonic with respect to processing time. Adding this constraint enables the process controlling the anytime algorithm (the proposed game agent in this case) to make a trade-off between the amount of time spent processing a problem, and the quality of the solution returned by the algorithm. If the guarantee of monotonicity is not added to the specification of an anytime algorithm, then the controlling process can have no idea when the optimum (or even appropriate) time to interrupt the algorithm is, as the quality could fluctuate. It is important to realise that although the anytime algorithm literature (e.g. [Zilberstein, 1996]) discusses the monotonicity constraint as a critical part of an algorithm, it could just as well be a feature of a complete anytime system, not just the underlying algorithm. For example, if the results of an anytime algorithm were accessed through an interface that always returned the best solution reached, ignoring the current solution being offered by the algorithm, then the whole system would fulfil the monotonicity criterion. Given this situation, it would still be preferable to have an underlying algorithm that tended towards monotonicity, as this would allow the overall quality of the returned results to improve more quickly.

The concept of an anytime algorithm can be illustrated with some examples. An example of something that is not an anytime algorithm (and could not be converted into one) is searching for an item. The only solutions to this problem involve finding the item, so halting the search before the item is found will not return a solution to the problem. Therefore, this is not a viable anytime algorithm[1]. The planning of a route between two known points makes a good example of an anytime algorithm (one that is used in [Zilberstein and Russell, 1993]). An initial solution can be constructed by plotting a straight line between two points. This initial result can then be iteratively refined to better suit the search criteria (e.g. to avoid crossing rivers, or to minimise the amount of energy used climbing hills). This approach is an anytime algorithm because a solution is available at anytime, and some heuristic function could be used to guarantee that any alteration of the route will obey the monotonicity constraint. An example of a path planning algorithm that suits this approach is presented in [Logan and Poli, 1996].

4.2 Why use an Anytime Planning Algorithm?

The flexibility that anytime algorithms provide make them an obvious choice for constructing complex algorithms that must function in real-time. This leads to one of the primary tenets of this thesis: in order to produce a planner that will function adequately as part of an agent in a real-time, dynamic and complex world, the planning algorithm used should be designed as an anytime algorithm. This section provides arguments in favour of this approach. The arguments are broken down into two sections; benefits that will be gained by an agent using such a planner, and benefits that will be gained by a game developer creating an agent to use such a planner. A number of these arguments were previously presented in [Hawes, 2002].

Before considering these arguments, it may be worthwhile to examine the bigger picture. Humans make plans all the time and are often constrained in the amount of time they have to produce them. For example, many sports involve producing plans in high-pressure situations, and the same can be said for many different professions. When making plans, humans appear to be able to reason at different speeds, at different levels of details, using different representations and, importantly, tend to be able to produce a finished plan whenever required. Although it is very hard to postulate systems that can reason in such ways (although they will certainly be far removed from the planners presented in this thesis), one of the benefits of pursuing research into anytime planning is that it may start to allow us to at least understand the issues that potentially shape and affect a human's ability to reason under pressure.

[1] Parts of this process could be viewed as an anytime algorithm. For example, although you may not have discovered the whereabouts of the item, as processing time increases, the information about where the item *is not* increases monotonically with respect to time.

4.2.1 Agent benefits

The arguments presented in this section are intended to highlight the benefits that a hypothetical agent using an anytime planner would gain over an agent using a traditional planner. The term "benefit" is used to refer to anything that enables an agent to function better. For example, an increase in the number of goals that can be achieved by an agent, or a reduction in the resources (time, energy etc.) used by an agent in achieving a particular goal.

Initially, an agent using an anytime planner gains the advantage of being able to explicitly manage resource usage during planning. A resource is anything that is consumed by the planning process. Typically this will be time, memory, or possibly energy (as thinking can be a difficult task). If the utility of achieving a goal decreases as time increases (e.g. the goal of plugging a hole in a container of liquid that is leaking steadily), then the utility of the goal can also be considered a resource when planning for such (time-dependent) goals. By using an anytime planner, the agent can trade off the consumption of resources during planning against the quality of the solution produced. This can be done by monitoring the progress of the anytime planner, either through direct feedback or via a performance profile, and then interrupting the planner when sufficient resources have been expended on creating a plan that surpasses a certain quality threshold. A performance profile is a representation of the relationship between processing time and result quality for a particular anytime algorithm and problem. Performance profiles can be used to predict how quickly a solution of a certain quality will be produced. The idea of performance profiles for anytime algorithms was first proposed in [Dean and Boddy, 1988] and has been extended in works such as [Zilberstein and Russell, 1996] and [ten Teije and van Harmelen, 2000].

The second advantage gained by an agent using an anytime algorithm-based planner is the ability to respond immediately to environmental change. Given that we have already considered that computer game environments are highly dynamic (see Section 2.3.3), and that we have seen how this relates to the problems inherent in planning in general (see Sections 2.3.3 and 3.1) it is this benefit of using an anytime planner that is critical to arguments in its favour. To an agent, the ability to interrupt its planner to retrieve a solution at any time provides a number of benefits. First, it means that should the agent decide that the current planning process is unnecessary (due, for example, to serendipitous achievement of the goal, or a change in agent motivation), it can simply terminate the planning process. This ability is probably available to all planners. Second, if the agent is able to predict when the environment will change in a way that is likely to affect its planning process, it can leave the planning process running until a certain amount of time before change is about to occur (e.g. the amount of time required to execute the plan), then retrieve and execute the plan before the change occurs.

The final advantage gained by an agent using an anytime planner is that anytime algorithms are ideally suited for integration into agent architectures. This is due to the inherent flexibility that is associated with interruptibility, and the additional information provided about the algorithm through either feedback or performance profiles. For example, an anytime planner can be interrupted, suspended and resumed. Therefore, the processing of an anytime planner can easily be interleaved with other tasks within an agent architecture, and other components in the architecture can easily monitor its performance. It is this last advantage that gives anytime algorithm-based planners an advantage over planners that are designed purely for speed (e.g. those discussed in Section 3.2.1).

These abilities, combined with a supporting agent architecture should provide the necessary power and flexibility for an agent to plan in a real-time, complex, dynamic environment.

4.2.2 Developer benefits

Because one of the aims of this thesis is to produce an agent that will behave intelligently in a computer game world, it makes sense to ask what advantages equipping an agent with an anytime planner will offer a games developer.

First, having a process that can be safely interrupted (and resumed) allows planning to be easily scheduled or spread across the available processing time. This could be achieved, for example, through the use of an AI process manager [Wright and Marshall, 2000]. Such scheduling allows efficient use to be made of the limited processing power available on all gaming platforms. Combining such a scheduler with an anytime planner would allow the quality of plans produced by an anytime planner to be automatically traded for processing time in order to achieve a constant frame-rate in a game. Without the anytime planner, reducing processing time for an agent through scheduling would result in a longer planning time. With anytime planning the planning time is the same as it would be without the scheduling, but the resulting plan is worse.

Second, being able to specify a flexible bound on planning time (by forcing an interruption to a planning process to occur after a certain amount of time), provides a way of implementing dynamic level of detail AI (e.g. [MacNamee et al., 2002]). Dynamic level of detail is a concept that was originally introduced by researchers investigating topics including graphics (e.g. [Luebke and Erikson, 1997]) and simulation (e.g. [Carlson and Hodgins, 1997]). Its basic premise is that more processing power is dedicated to objects that are more significant to the players experience, and less processing power is dedicated to less significant objects. In graphics terms, this means rendering objects close to the observer in greater detail than those objects that are more distant. In AI terms, dynamic level of detail means spending more CPU cycles determining the behaviour of the agents that the player is interacting with, and less on those that are not currently being interacted with or are not significant to the player's experience. To do this using an anytime planner would require the following behaviour. For distant or relatively insignificant agents, an early interruption of an anytime planner would result in a plan that will (depending on how the agent's domain is encoded) guide the agent to achieve its goal in a basic (possibly clumsy or inefficient) way, after minimal processing. For agents that are significant to the player's experience (either due to being closer or because their actions will have a greater impact on the course of the game) an anytime planner can be run for a lot longer, resulting in more detailed and hence more efficient (or interesting, believable, faster etc.) behaviour when the plan is executed.

Third, the development work involved when working with a planning-type system is often a lot less than is required with a reactive system. In the first case the developer has to implement the planner and planning domain description. In the second case the developer must write all the rules to determine the behaviour of the agent being implemented. Specifying all the agent's behaviour by hand may often take longer than allowing the plans to be automatically generated at run-time. Once a domain independent planning algorithm has been implemented, the behaviour of the planning agent can be altered by changing the description of the planning domain. Such alterations typically produce predictable changes in agent behaviour. In contrast, changing behaviour by amending reactive rules often leads to unanticipated behaviour as rules can interact in unpredicted ways (such an occurrence becomes more likely as the number of rules increases), although this behaviour can be debugged by examining individual rules. Also, because most planners are domain independent, a planner implemented for one game or agent can be reused for another game or agent, even if the domain requirements are significantly different[2]. This is unlikely to be the case for the reactive systems currently being used in game agent development.

A final advantage gained by the developer using an anytime planner is an agent that can display "believable" weaknesses. Humans do not always form correct, optimal plans when under pressure (when playing computer games or at other times), so competing against an agent that always formed correct plans in all situations would enforce the feeling of interacting with something artificial (although a "perfect" style of play would be useful for certain game characters e.g. robots or super-beings). An agent executing plans returned from an interrupted anytime planner would almost certainly not always display such "optimal behaviour", and could possibly fail to achieve its goals altogether. Based on this, an upper bound on planning time could be used to characterise certain styles of gameplay and agent behaviour. Such playing styles could range from considered and cautious (a high upper bound, therefore more planning time) to gung-ho and carefree (a low upper bound, therefore less planning time).

4.3 Possible Problems with Anytime Planning

The previous section offered arguments for why a planner based upon an anytime algorithm would be beneficial for both agents and agent developers. This section presents some arguments to the contrary, and possible refutations of these arguments.

The first argument that can be levelled against anytime planning is that if a fast enough planner implementation could be developed, then planning solutions could be generated before the need to interrupt has arisen. This can be countered by arguing that an anytime planner could always be used to tackle problems with greater complexity than a non-anytime planner, as it would not necessarily be run to completion and would therefore not be as vulnerable to the scalability issues that accompany an increase in domain complexity (although this argument is entirely dependent on the way in which the anytime planner is designed). A second counter argument is that properties of an anytime algorithm will always be useful for when the agent is tackling unexpected situations that stretch its resources to its limits, or if multiple planning problems need to be tackled concurrently (assuming that the agent would not have enough time or other resources to do this in a non-anytime manner). This view is backed up in [Howe et al., 1990] in which they identify anytime processing as a suitable approach for problem solving in a real-time agent.

[2]This would not be the case if the planner in question was domain dependant.

A second argument that can be presented against the use of anytime algorithms is that early interruptions may provide answers that can barely be classified as solutions to the problem being tackled. A good example of this can be taken from the path planning anytime algorithm described in Section 4.1. If it is interrupted early, the solution returned will be a straight line between two points. This solution will have been created by an initial guess and will have not taken into any account of the constraints of the search. The justification for this behaviour is that there will always be times when any planning system (e.g. human, robot or software) will be short of time to complete a plan. In such a situation, an anytime planner may be more useful than any other system as it is specifically designed for such conditions. Given a situation in which an anytime planner would be struggling for processing time, almost any other comparable method of processing would also be struggling for processing time. Also, other methods would not fail as gracefully as an anytime planner (as anytime planning still produces a solution in this situation). Some existing anytime planners (to be reviewed in Section 4.4) solve the problem of returning potentially poor solutions after a very short of time by preventing interruptions occurring before an initial complete solution has been found. This stops non-solutions from being returned, but prevents the planner from being useful when only very short periods of time are available (i.e. periods shorter than their minimum run time). It is demonstrated later in this thesis (see Chapter 9) that the nature of the proposed anytime planner (in Chapter 5) enables the planning agent to successfully execute even plans created in the early stages of the planning process. In the general case, the behaviour generated from such plans should allow the agent to achieve its goals, but in a manner that is of a lower quality than if the planner had been allowed a longer time to run.

A third argument against the use of anytime algorithms is that the amount of additional work above and beyond that required to design, implement and run a non-anytime planner is prohibitive. Although both the development of the anytime planner and the agent architecture in which the planner is eventually embedded have required developments beyond those normally required for a planner and agent combination, the end result of the work is an agent that is demonstrably more powerful than a "standard" agent (i.e. one without anytime capabilities) (see Chapter 9).

The final argument against using an anytime planning algorithm is the challenge that it might not actually be possible to convert a standard planning algorithm into an anytime algorithm. This argument stems from the fact that we have earlier (in Section 4.1) demonstrated that it is impossible to turn some algorithms (e.g. searching for an object) into anytime algorithms. It is the position of this thesis that it *is* possible to convert a particular planning algorithm into an anytime algorithm that operates based on certain assumptions (e.g. what constitutes a solution to a planning problem for agent behaviour). Anytime planners have also been successfully developed by other researchers. Their work is presented in Section 4.4.2.

4.4 A Review of Anytime Algorithm Literature

This section presents previous work related to anytime algorithms. Section 4.4.1 covers two proposed frameworks for developing anytime algorithms from knowledge-based algorithms. Section 4.4.2 then covers the existing work on developing anytime planners.

4.4.1 Anytime algorithm frameworks

In [Hansen et al., 1997] the authors propose a technique for converting heuristic search algorithms into anytime algorithms. This is of interest because planning is essentially a search problem. The root node of a planning search tree is the initial state, and subsequent search nodes represent subsequent planning states. Successor search nodes are generated by adding the effects of any applicable operator to the previous state. These basic principles can be altered for different planning approaches (e.g. backward-chaining planning), but the search framework does not change significantly.

Hansen et al.'s framework for converting heuristic search algorithms to anytime algorithms requires two changes to the standard A* search framework[3]. First, a non-admissible heuristic is used to guide the search, and second, searching continues after the first solution has been found. The effects of these changes are as follows; the non-admissible heuristic causes the search to quickly find a non-optimal solution, then search is allowed to continue to find solutions that are closer to optimal (eventually converging to the optimal solution). The intuition behind this behaviour is that it is important to find a solution as quickly as possible in case an interruption occurs, then this and subsequent solutions will be available if an interruption occurs whilst searching for the optimal solution.

A non-admissible heuristic is generated using the idea of weighted A* (WA*), which gives increased significance to the estimated cost of the optimal solution [Pearl, 1984, Chapter 3]. When the estimated cost is weighted lightly, the search process performs more like a breadth-first search, exploring multiple branches without committing to one particular branch and therefore taking a longer time to find the shortest possible plan. When the estimated cost is weighted heavily, A* performs more like best first search and heads directly towards possible solutions without taking search path length (i.e. plan length) into account. An admissible heuristic never over-estimates the heuristic cost of possible solutions, and maintains an appropriate balance between finding solutions and minimising search depth. Weighting the heuristic cost in WA* therefore encourages non-optimal solutions to be found earlier than optimal solutions would be found by A*. Consequently, this weighting also dictates the shape of the performance profile of the anytime algorithm.

As a method for creating anytime algorithms from search problems, the ideas presented in [Hansen et al., 1997] represent a good method for extending existing search technology that complements the requirements of anytime algorithms. Unfortunately, this approach to creating anytime algorithms fails to make the resulting algorithm interruptible at *any* time. During the period before the first solution is found there is no solution available, so an agent using a planner based on this method will not be able to interrupt the planning process and retrieve a solution before this point. This may not be a problem in some domains (i.e. those with shallow non-optimal solutions), but in others it is possible that even the solutions that are easiest to find are still deep within the search space and require a not

[3]Explanations of A* and the other approaches to search mentioned in this section can be found in Chapters 3 and 4 of [Russell and Norvig, 1995].

insignificant amount of time to find.

Another general approach for creating anytime algorithms is presented in [Mouaddib and Zilberstein, 1995]. The authors have focused their research on creating knowledge-based anytime algorithms. This focus was chosen because knowledge-based algorithms (e.g. planning or symbolic natural language processing) are powerful reasoning techniques, but have unpredictable performance profiles. This unpredictability means that it is harder to construct informative performance profiles for such algorithms.

Mouaddib and Zilberstein's framework for creating knowledge-based anytime algorithms exploits the concept of abstraction (as presented in Section 3.1). Processing is carried out at one level of abstraction until a level of sufficient solution quality is reached. Once this level has been reached, processing moves on to a more fine-grained description of the problem. This process continues until either an interruption occurs, the amount of processing time is exhausted, or a solution of sufficient quality and detail is reached.

The advantage of processing in such a manner is that by structuring the way in which complexity is introduced, the performance profile of the algorithm becomes more predictable. Such a processing style also allows larger (i.e. more abstract) chunks of problems to be solved early in the search process, with later processing fine-tuning the results. This gives anytime algorithms based on this framework the behaviour of offering diminishing returns, a behaviour that has been cited as desirable for an anytime algorithm (although not strictly necessary) [Zilberstein, 1996].

The progressive processing framework for constructing anytime algorithms from knowledge-based algorithms relies on two important assumptions. First it requires that a problem can be represented in a manner that has distinct levels of abstraction, and that a representation from one level of abstraction can be used to guide processing at the next level of abstraction. The second assumption is that solutions that are not completely primitive (i.e. still abstract) are acceptable as solutions to the problems being solved by the anytime algorithm.

These assumptions are critical to being able to produce a planning anytime algorithm. As was presented in Section 3.1, planning problems can be represented and solved using various abstraction-based methods. Of the available abstraction approaches, HTN style planning is not particularly suited to this style of processing, as the discrete levels of abstraction required by the previously mentioned framework are not always present in the domain description. Precondition removal planning is better suited to the framework, as discrete levels of abstraction are present because the addition of preconditions to the planning problem happens in fixed sets. Unfortunately, the second assumption (that abstract solutions can be returned as full solutions) could possibly be violated by precondition removal planning. If certain preconditions are not considered, then related actions will not be added to the plan. If these related actions are necessary to achieve the goal that was planned for, then the plan returned by this method is not a solution, and the assumption made by the framework is violated. If the planning agent had the ability to repair plans whilst acting them out, then this violation could be removed as the plans could be altered as necessary. This would leave the number of actions missing from the plan as a good quality measure for the anytime planner. A planner with many missing actions would require a greater amount of repair, and would therefore be a lower quality plan than one requiring less repair. The idea of using a measure of what is missing from an abstract representation of plan as a quality measure is revisited in Section 5.2.

4.4.2 Anytime planners

In this section a number of existing anytime planners are reviewed and their behaviours are compared to the requirements for anytime algorithms.

The first anytime planner we will look at is an altered version of the Prodigy planner that is used in an agent architecture from the Oz project [Blythe and Reilly, 1993]. Unfortunately, the anytime algorithm hasn't been fully implemented, and can only operate with a fixed time limit (a contract algorithm). The planner is called whenever a novel plan is needed by the Oz agent. This call is accompanied with a time limit for planning. A problem with this anytime planner is the manner in which Prodigy has been altered to become an anytime algorithm. Rather than alter the actual planning algorithm, the way in which the planner decomposes conjunctions of goals has been altered. Instead of working on the whole problem in an anytime fashion, the planner is called to plan for separate parts of the problem one after the other. Then, if enough time remains, combinations of the parts are planned for. An example of this is if the planner is passed a conjunction of goals $g1 \wedge g2$, it will first plan for $g1$, then $g2$ (the actual order is decided by a priority value), and finally $g1 \wedge g2$. The advantage of this is that it gives the planner small chunks of work to do, and will hence have an answer earlier. This is necessary because the planner can only return complete plans (no partial solutions). If the planner fails to return a plan then the agent has nothing to do. Although the approach of separating a plan's goals is a workable solution, it has three flaws. The first is that the early plans may miss better solutions that involve the interactions of the subgoals. The second flaw is that there may be situations in which plans for a single goal from conjunction may prevent another goal from being planning for unless they are planned for simultaneously. The final flaw is that an early interruption to the planner may return a plan that does not cover a number of the goals from the initial conjunction. This will leave the agent unable to achieve its goal.

The next planner to be reviewed is APS, a Prolog-based anytime planning system [Prendinger and Ishizuka, 1998]. The work is based on the concept of planning as theorem proving (e.g. [Green, 1969]). APS handles the possibility that planning can be interrupted by allowing a wider class of solution to be returned from a planner. In addition to standard solutions (i.e. theorems/plans with all their constraints satisfied) the APS planner is allowed to return those that rely on default assumptions that are unproven (the authors call plans based on these assumptions "approximate plans"). Default assumptions are the `not` preconditions of planning operations (e.g. that the block is `not` on the table) where the "not" is interpreted as "negation-as-failure". The result of this is that given "favourable conditions" an approximate plan can be executed even without these preconditions being met. By "favourable conditions" the authors mean conditions that do not clobber the action for which the `not` preconditions are being ignored (i.e. as long as the block being on the table does not prevent any subsequent actions, then it's safe).

This anytime planning behaviour is placed within a mechanism that manages the plans created and handles any interruptions. This mechanism functions in the following way. First, planning proceeds until an approximate plan has been completed, and the deadline for plan generation has not been exceeded. Then, until the deadline is reached, one of two operations is performed on the approximate plan. If the plan is consistent with the environment (i.e. it could be executed successfully), then further work goes into computing the approximate plan. If the plan is inconsistent with the environment (i.e. it would not be possible to execute it successfully) then either an attempt is made to repair the plan (to create a new approximate plan) or, if immediate action is required by the planning agent, it employs its emergency module. The details of the emergency module are not specified in the literature, but it appears to be some form of reactive planning system.

The crucial problem that makes the APS system unsuitable for generating behaviour for an agent in a computer game-like environment is that it is not truly an anytime algorithm. The processing mechanism described above does not allow for unpredicted interruptions, as all processing is done with reference to a limit on processing time. An interruption before this predicted point could fail to provide a solution at all. An additional problem is caused by having to initially find an approximate plan before any solution can be returned. Although the authors claim that this will be a fast process, it is possible to imagine a situation in which this process requires more time than is actually available for the planning process (e.g. on a slow machine). On this issue APS is similar to the previously reviewed framework for transforming heuristic search algorithms into anytime algorithms. A final problem with the APS method of anytime planning is that without the necessary "favourable conditions" any approximate plan generated by the system will fail (a fact that is outside of the agent's control). This is a fact that all anytime planners will have to deal with. Because the anytime paradigm allows for processing to be interrupted before enough time has passed to find a solution using a regular algorithm, the returned plan will always be flawed in some way. The flaws may differ between planning algorithms and representations, e.g. missing concrete steps in the solution (as was hypothesised for precondition removal planning) or a lack of detail because of the level of representation being used (as in the path planning example in [Zilberstein and Russell, 1993]), but they will always be present. Flaws may also vary on their level of severity. For example, a seriously flawed plan may miss all but its start and end actions, whereas an almost flawless plan may have all the correct planning actions, but may have some constraints yet to be resolved in order to ensure it is a correct solution.

[Shoaff et al., 1994] present an anytime planning algorithm based on a plan representation that differs greatly from the STRIPS-style representations that we have dealt with so far. They represent plans as Markov Decision Processes (e.g. [Blythe, 1999]). Markov Decision Processes (MDPs) are constructed from states and transitions. States are usually used to represent a particular collection of properties. Transitions are ways of moving between states and are associated with a probability of being used. When planning using MDPs, primitives in a complete plan are represented as transient Markov states (those with a transition to another state) and end states (goals and failures) are represented as absorbing states (those with no transitions to other states). Plan quality (an essential part of creating an anytime algorithm, see Section 5.2) is represented as the probability of reaching the goal state from the current point in the plan (which corresponds to a particular state in the Markov chain). This approach allows plan quality to be revised at runtime as moving through the Markov chain alters the probability of ending up in a particular goal state.

Because of this different representation, the process of developing a plan is considerably different from the planning methods presented previously. Before a plan can be developed, a "plan embryo" must be constructed. This contains the start task and all the possible Markov models of the world based on the initial state (i.e. Markov chains informing the planner about the probabilities of state transitions). This embryo is then passed to the planning process proper. To generate a plan, the current Markov state (selected in a breadth-first manner) is examined, and then the task which represents the maximally probable transition from the current state is added to the plan. Once a plan has been completed, the process starts from the initial state again, this time considering the next maximally probable transitions. The current best plan (i.e. the one with the highest probability of achieving the goal given the initial state) is stored and is returned if the process is interrupted.

Although this process provides the planning agent with a great deal of flexibility (including the ability to switch between plans as quality changes as more information is gathered at runtime), there are a few questions over its anytime behaviour. First, the construction of the "plan embryo" is a time consuming process that cannot be interrupted. It is time consuming because all possible transition chains must be constructed from the initial state, and there is a potentially a large (and exponentially increasing) number of them. It cannot be interrupted because all subsequent planning operations rely on the embryo. To overcome the problems presented by the embryo construction phase, the authors propose to calculate all the transition models for a particular domain offline. This would certainly be necessary unless the domain model could not be calculated until runtime (e.g. if the planning agent was placed into a hitherto unknown environment in which its normal actions have unknown effects). The second problem that occurs with this approach is the problem that we have seen occurring with other anytime planning methods; there is no plan to return before the first solution has been returned. The authors propose to deal with this by having default plans which can be used either in place of a deliberatively generated solution, or combined with a partially complete plan. The approach of combining default actions with generated actions is a flexible way to overcome the problem as it adds the power of dynamically generated actions to precompiled plans. The only problem (one noted by the authors) is that there is no guarantee that the default plan leads to a solution in the current situation, and this could lead to a drop in the performance of the planning agent (cf. the use of plan schemas in [Atkin and Cohen, 1998], reviewed in Section 3.2.3).

The next anytime planner to be reviewed is the planner developed in [Briggs and Cook, 1999]. This anytime planner combines a number of previously reviewed technologies into one system. The basic planner is an abstraction-based precondition-removal planner. This planner is used by the system to generate a plan until an interruption occurs. As presented previously (see Section 3.1), this approach to planning creates plans at a primitive level but ignores certain preconditions of primitives earlier in the planning process. As planning proceeds, more preconditions are considered until the plan is complete. If an interruption occurs before the plan has been completed there will potentially be primitives missing from the plan (primitives that would be added later to satisfy currently ignored preconditions). To overcome this problem, the partial plan is passed on to a plan completion component to finish the planning process. Briggs and Cook's anytime planner uses PRS (see Section 3.2.2) as the plan completion component. PRS is a reactive planner and so it is a potentially fast method of completing plans.

The use of PRS to complete partial plans is both the main strength and main weakness of this approach to anytime planning. By having a plan completion step that occurs after an interrupt, the planner ensures that all plans that are returned are correct and complete. By making this process reactive, the authors aim to make it as fast as possible. The reactive plan completion step allows the user of the anytime planner to trade-off the amount of deliberative planning against the amount of reactive planning by choosing when to interrupt the planner. A late interruption will take advantage of a longer period of deliberative planning (and the advantages this approach offers), whilst an early interruption passes more of the responsibility of plan generation to the reactive planner.

The problem with having a plan completion stage is that it will potentially introduce a delay between the interruption of the planner and the subsequent return of a plan. Because the completion process is reactive, it should be faster than any deliberative completion process, but because it is still a planning process it could still potentially take a long time. The actual time taken largely depends on the complexity of the planning problem and the amount of rules used in the reactive planner. The harder the problem is to solve, the longer the completion process will take. An early interruption to this anytime planner will result in a harder planning problem than a later interruption (because less of the plan will have been specified), resulting in a longer completion phase. This is a problem because an early interruption to the planning process usually means that the controlling process is under greater pressure than if a later interruption occurred, and it therefore requires a solution as soon as possible. In general terms, any potentially significant delay between an interruption and the return of a plan is undesirable because the controlling process has interrupted the planner to get a result, and any delay may decrease the utility of the result with the respect to the predictions of the controlling process.

The final anytime planning system to be reviewed is also the most important as far as this thesis is concerned. The Excalibur project has been developing an anytime planning system with almost identical aims as those of this thesis; to generate behaviour for an agent in a computer game-like world. The planning process is based on solving structural constraint satisfaction problems through local search [Nareyek, 2000].

Structural constraint satisfaction problems (SCSPs) are an extension of constraint satisfaction problems. In constraint satisfaction problems, a parameterised problem structure (in this case a plan) has its parameters (variable assignments in the case of planning) refined until some set of constraints over the assignments are satisfied. To plan in this manner, an initial plan structure must be present which then has its variables altered, whilst its structure remains static. The inability of constraint satisfaction to alter plan structure (constraints can only affect variable assignments) makes it unsuitable for planning for problems where pre-built plan structures cannot be used (e.g. highly dynamic domains)[4]. SCSPs extend the constraint satisfaction problem paradigm by adding operators that can change the structure of plans

[4]To overcome this limitation some constraint satisfaction planners start with a maximal plan containing all possible actions, but this seems inefficient for problems with even moderate branching factors.

(or whatever representation is being manipulated) in addition to changing their variable instantiations. To ensure that the plan is kept consistent, plan structures have constraints associated with them in a similar manner to plan variables.

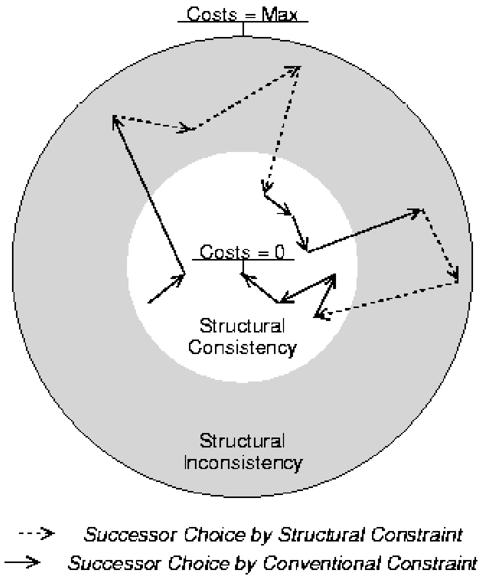

Figure 4.1: How SCSP creates plans [Nareyek, 1999].

SCSP plans are generated through a process of iteratively changing the plan structure, starting from an initial state [Nareyek, 1999]. This usually means starting with a particular plan structure and then altering its variable assignments in order to satisfy some constraints on it. This may lead to the alteration of the plan structure to satisfy a structural constraint, or further alterations to the variable assignments. This process of plan generation using SCSPs is demonstrated in Figure 4.1. Plan quality in SCSP plans is measured by how inconsistent the current plan is, and is based on both structural and variable constraints. Changes to plan structure are chosen by selecting which alteration offers the greatest reduction in plan inconsistency, with weightings being applied to different constraints based on their importance. Changes can be made to variables or plan structure at any time, with the only preference between the types of changes being based on how they improve plan cost. In addition to basic action planning, the SCSP planning model supports temporal reasoning, real valued resources and planning with incomplete knowledge.

The SCSP planner is offered as an anytime planner, and it does have the basic properties required for anytime behaviour. Because it iteratively alters the plan via local search through the space of fully instantiated (although not necessarily correct) plans, a solution is always available at any time [Nareyek, 2001]. Although the planning representation and process employed by the SCSP method are very powerful, there are some problems with its anytime behaviour. The inconsistency measure used as the basis for plan quality does show a general macroscopic increase (i.e. the plan quality increases in general), but when examining performance profiles from a case study in [Nareyek, 2001] it can be seen that monotonicity is not guaranteed between adjacent iterations of the planning process. This problem could be trivially overcome by storing the lowest cost (most consistent) plan for return should an interruption occur. A more critical problem for anytime behaviour generation using SCSPs is caused by the very nature of constraint satisfaction problems. Because the plan being developed remains inconsistent throughout almost all of the planning process (either in terms of structural or conventional constraints), there is a possibility that the resulting plan may not be executable by the planning agent due to its inconsistencies. A less serious case may see an agent violating resource usage or soft temporal constraints imposed upon it. This latter case is arguably allowable in the domain of computer games as we are less interested in correct solutions than we are in generating entertaining behaviour (we are not taking a solution-oriented approach). Unfortunately, if constraints more central to the plan's execution (e.g. remembering to open a door, or picking up your backpack before you attempt to deliver its contents somewhere) are violated, then no meaningful behaviour can be generated from such a plan. A way to tackle such a problem is to heavily weight the inconsistency values associated with these constraints so that they are dealt with as early in the planning process as possible.

4.5 General Observations on Anytime Planning

Most of the anytime planners and anytime frameworks that have been presented have common features. These features are some form of initialisation period, followed by an interruptible processing phase. In addition to this, each algorithm usually processes problems with reference to a class of solutions that they are allowed to return.

The initialisation phase typically involves some form of obligatory processing, the result of which forms the starting point of the interruptible processing phase. We have seen examples of a variety of this kind of preprocessing. The path planning example and SCSP use a best guess at an initial state which is subsequently elaborated. APS and the framework for anytime heuristic search extend this guess into some costly processing by searching for either a shallow initial solution, or something approximating a solution. Finally, the Markov Decision Process-based anytime planner requires an embryo structure to be built from which the plan is extracted. Selecting between the first two of these approaches requires a trade-off to be made. By starting with a quick guess, processing can be interrupted almost immediately, but the early results may be very poor. By extending the initial processing phase, the period before an interruption can occur is also extended, but when interruptions do occur, the results will be of a higher quality. The approach of building a separate structure from which a plan is gradually extracted appears to inherit the worst of both of these approaches. There is an extended initialisation period which prevents interrupts, and then the processing phase does not have a solution to return immediately.

The only approach that does not fit neatly into this classification of initialisation phases is the anytime planning system developed by Briggs and Cook [Briggs and Cook, 1999]. As the planning process is based on a generative planner, the only initial processing is the construction of an initial state (something that all approaches must do and that is not really part of the planning process). If this anytime planner is interrupted immediately after starting processing, its plan completion component (a reactive planner) must complete the plan. In this case, the plan completion phase is the obligatory processing, but it occurs after the planning process rather than before. Although any obligatory processing in an anytime algorithm is bad (as it reduces its flexibility), the approach of having obligatory processing occur after an interruption has one key advantage. Because the amount of plan completion required is dependent on the amount of planning performed previously (a more complete solution requires less work than a less complete one), the amount of obligatory processing is inversely proportional to the amount of processing time already invested in the problem. This means that the amount of obligatory processing reduces as the amount of interruptible processing increases (although this has already been identified as a potential weakness of this approach when planning time is limited).

The interruptible processing phase is similar across all anytime algorithms. During this phase the processing can be interrupted at any time and a result will be returned. To ensure that the algorithm is a useful anytime algorithm, the quality of this result should increase monotonically with respect to processing time.

Perhaps the most interesting of the points of comparison for anytime planners is the class of solution they allow, and what this may mean for an agent using them. The simplest examples are planners which only return solutions that would also be allowable solutions in the non-anytime versions of the algorithm. This style of anytime algorithm (e.g. the A*-based anytime framework, the MDP-based anytime planner and Briggs and Cook's planner) will always produce correct solutions, but trade this off against limited interruptibility (as the algorithm must wait for fully formed solutions to be found). Once an anytime algorithm allows a wider class of solution than this, there appear to be two different approaches that it can take.

The first of these is to allow any primitive solution that falls within a certain set of constraints. This approach includes allowing negation-as-failure solutions in APS and any solution from the SCSP approach. The advantage of this approach is that the resulting solutions are presented in the same (primitive) language as full solutions. The key disadvantage is that, depending on how "wide" the class of solutions is, these primitive plans may fail to achieve their goal if the assumptions used to generate them are not valid in the execution environment.

The second approach used to define a "wider" class of solution is to allow any valid solution specified at an abstracted level of detail. An example of this is the approach used by the progressive processing framework. The advantage of this approach is that it tends to allow interrupts to occur earlier into the search process (or at *any* time) as solutions to problems tend to be much easier to find at higher levels of abstraction. The principal disadvantage of this approach is that solutions specified in non-primitive representations may not be directly executable by the planning agents. This generates the need for an additional interpretation process (this view is expanded in Section 5.2).

These two approaches are analogous to the two approaches to abstraction typically used by planners. The primitive constrained solutions approach is similar to precondition removal abstraction (only a primitive representation is used, and early solutions are allowed to fall outside of the usual class of planning solutions). The abstract representation approach is similar to hierarchical task network planning (less detailed representations are used for earlier parts of the planning process).

4.6 Summary

This chapter introduced the concept of an anytime algorithm and presented the potential advantages and disadvantages of taking an anytime approach to planning. This led to a review of previous approaches that have been taken to the general task of converting certain existing types of algorithms to anytime algorithms, and the more specific task of creating anytime planning algorithms. The chapter closed by generalising from these previous approaches to identify some features common to all anytime planning approaches. This enabled a discussion of the strengths and weaknesses of the potential variations of these more general features (e.g. the trade-offs of having a prolonged initialisation period for an anytime planner).

Chapter 5

The A-UMCP Anytime Planner

This chapter introduces one of the major contributions of this thesis, the Anytime Universal Method Composition Planner (A-UMCP). A-UMCP overcomes some of the weaknesses of the previously reviewed anytime planners (most notably the lack of true anytime behaviour), whilst adhering to the principles of the paradigm. A-UMCP is based on hierarchical task network planning and a combination of informed and uninformed search using a quality measure described below as a search heuristic.

Two of the principal issues when designing an anytime algorithm are ensuring that it is interruptible, and developing a suitable quality measure for it. The following two sections discuss why these issues are important and how they relate to planning. Both sections then present the ways these problems have been solved in the developed anytime planner. Following that, the algorithm used to generate the quality measure used in A-UMCP, and the search framework used to explore the space of possible plans are presented.

5.1 The Interruptibility of Planners

The most powerful feature of an anytime algorithm is that it can be interrupted at any time, and still return a "usable" solution. The reasons why this is a desirable feature of a planning algorithm have already been presented in Section 4.2. To be as useful as possible, the anytime planner must be interruptible at *any* time, not just once a particular point has been passed in the processing (e.g. the first solution has been found). It is this feature of an anytime algorithm that the previously reviewed anytime planning algorithms (e.g. APS [Prendinger and Ishizuka, 1998] and the Anytime Heuristic Search framework [Hansen et al., 1997]) have had problems implementing.

There are two reasons why the existing anytime planning algorithms have failed to thoroughly capture this important feature of the anytime paradigm. First, there are those planners which require a complex structure to be built from which a plan is then extracted. This is the way the anytime planner presented in [Shoaff et al., 1994] works (as described in Section 4.4.2), and non-anytime planners such as GraphPlan [Blum and Furst, 1997]. The hard work of planning (e.g. resolving inconsistencies between actions) is usually done during the generation of this initial structure, and then the extraction of a plan is usually a simpler process based upon the initial state given for the problem, or other restrictions on the desired solution. Anytime planners in this vein cannot be interrupted before their initial planning structures have been built. Although the details will vary between planners, interruptions during the extraction phase will be subject to the same constraints as interrupting a classical planner (e.g. that interruptions must leave a completely extracted solution).

The second reason why anytime planners have failed to implement complete interruptibility is due to the way state-based (i.e. non-abstraction based) planners construct plans. As was discussed in Section 3.1, state-based planners construct plans by adding one action at a time, until a complete and consistent plan has been produced. Because the consistency of the plan is affected by every action added, it cannot usually be determined whether a plan will be consistent (i.e. whether it will achieve its goal) until the final action has been added to it. Because any result returned by an anytime planner must be a complete plan (i.e. its final action must be added to it to ensure that it is valid), this results in a lack of interruptibility in state-based planners (and the same for the extraction of plans from Markov models or planning graphs).

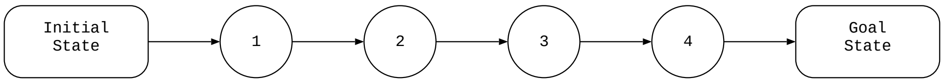

Figure 5.1: A complete plan created by a state-based planner.

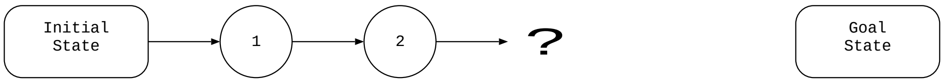

Figure 5.2: An interruption to a forward-chaining state-based planner.

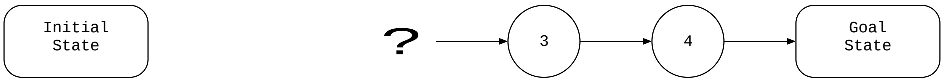

Figure 5.3: An interruption to a backward-chaining state-based planner.

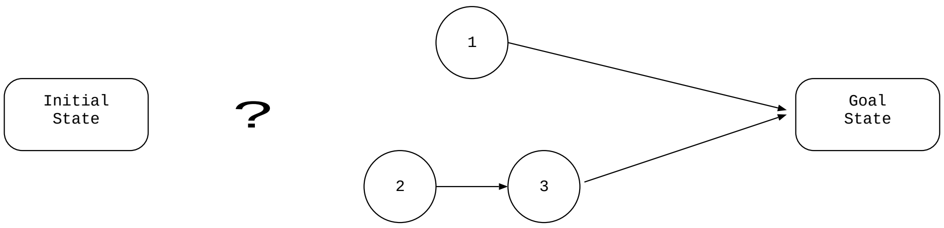

Figure 5.4: An interruption to a partial-order state-based planner.

To illustrate this point we can examine the results of interrupting a variety of planning processes. Figure 5.1 is a graphical depiction of a plan, with five operators (the arrows in the diagram) taking the planning agent from an initial state to a goal state via four intermediate states. Figure 5.2 depicts what happens when you interrupt the forward-chaining planning algorithm that would produce the plan shown in Figure 5.1. The result is a *plan fragment* (a part of a plan that is consistent in itself, but does not span between the initial and goal states in the planning problem) that leads from the initial state, to some unknown place in the state space. Although the planning agent would be perfectly able to execute this plan fragment, the agent has no guarantee that the goal state is reachable from the end of the plan fragment, or that the end of the plan fragment is even any closer (in terms of the number of actions required to bridge the space between the states) to the goal state than the initial state is. This is because the planning process has not yet bridged the gap between the end of the plan fragment and goal state. The plan fragment could actually take the agent further away from its goal. A similar situation occurs after interrupting a backward-chaining planner (Figure 5.3). The remaining plan fragment is guaranteed to take the agent to the goal state, but only if the agent can reach the state that represents the start of the plan fragment (state three in Figure 5.3). Again, because the planning process has not yet successfully bridged the gaps between the initial state and the start of the fragment, the planning agent has no guarantee that the start of the plan fragment is not further away than the goal itself, or is even reachable by the agent. The final example is in Figure 5.4, and it demonstrates what happens when an interruption occurs during a partial-order planning process. Instead of the one plan fragment that was left after the interruption in the previous examples, an interruption to a partial-order planning process will leave numerous plan fragments. These fragments are the same as those produced by interruptions to other state-based planning algorithms, therefore the planning agent has no guarantee that these plan fragments can be connected to produce a plan that will lead to its goal.

To generalise from the previous examples, we can state that if plans are built action by action, an interruption will leave behind plan fragments that whilst internally consistent are not guaranteed to be part of a plan between the initial state and the goal state. To be able to return a solution at any time during the planning process, a method that operates on the entire plan at once (not action by action) is needed. A local search method combined with a constraint satisfaction problem would operate on a whole plan structure at once, but as was observed during the development of the SCSP anytime planner, the necessary use of maximal plan structures leads to complexity issues [Nareyek, 2000]. This is overcome in SCSP through the use of operators that can alter plan structure as well as variable assignment.

Another planning method that operates on the entire plan at once is hierarchical task network (HTN) planning [Tate, 1977]. As was described in Section 3.1, HTN planning is an abstraction-based planning technique. The whole plan is always represented during the planning process, subdivided into abstract tasks. The planning process involves replacing abstract tasks with less abstract tasks until no abstract tasks are left in the plan (when all the tasks have been reduced into primitive actions). Because of the way HTN planners produce plans, an interruption at any point during the planning process will return a complete plan, albeit one represented at varying levels of abstraction. The very start of an HTN planning problem is often just one very abstract action, but even this is a complete representation of the entire plan (although it does not contain much information). Because an HTN planner always has a representation of the complete plan available, it can be interrupted at any time. This reasoning has led to an HTN planner being chosen as the basis for the planner presented by this thesis. The intuition behind using HTN planning as the basis for the anytime planner is that any plan will contain some information to guide the agent to achieve its goals. A plan that has been processed for longer will contain more specific information than one that has had less planning time.

The relationship between anytime processing and abstraction is the basis of the progressive processing anytime framework [Mouaddib and Zilberstein, 1995], reviewed in Section 4.4.1. This relationship is so strong that it is difficult to postulate a planner that can be interrupted at any time to provide a complete solution that does not use abstraction. Most of the previously reviewed anytime approaches used abstraction in some form, whether through abstracted representations, or by abstracting away detail in the problem (by ignoring certain constraints or actions).

The major disadvantage of using an anytime HTN planner for generating agent behaviour is that, by their very nature, the abstract tasks that make up HTN plans are not directly executable by the planning agent. In all planning problems it is only the action primitives that can be executed, and only when a solution is found in HTN planning is the plan completely composed of primitives. If we were interested in an anytime planner solely for completely solving problems, then this would prove a distinct barrier to the application of this method. Because we are interested in a planner to form part of an agent architecture (concerning ourselves with agent-oriented rather than solution-oriented planning) we can pass the problem of dealing with the abstract plan representation to another component in the architecture. This component is described in Section 8.4.1.2.

To create an anytime HTN planner, it is first necessary to choose an existing HTN planner on which to base the algorithm. Although greater flexibility could be gained through developing everything from scratch, constructing an anytime planner on top of an existing algorithm minimises the amount of initial work required before the anytime features are added. There are numerous existing HTN planners that could be selected. For example, SHOP [Nau et al., 1999], SIPE [Wilkins, 1988], or O-Plan [Currie and Tate, 1991]. The HTN planner that was chosen to provide the planning basis of the anytime planner was the Universal Method Composition Planner (UMCP) [Erol, 1995]. The planner presented by this thesis is named the Anytime Universal Method Composition Planner (A-UMCP for short) because it is an anytime version of the Universal Method Composition Planner.

There are a number of reasons why UMCP was chosen. First, its algorithms and representation are based on formal concepts and are well documented [Erol et al., 1994]. This could be said of some of the other HTN planners mentioned above. Second, UMCP is solely an action planning algorithm. Other HTN planners combine planning with execution and plan monitoring (e.g. SIPE and O-Plan), or can solve problems with numerical values (e.g. SHOP). Because we are interested in a planning algorithm to form a component in an architecture, we only want to focus on action planning and not additional execution features. Using numerical values during planning is a powerful feature, but represents an added complication when attempting to develop a novel anytime planning algorithm. The final feature of UMCP that caused it to be chosen is that it uses a powerful and flexible representation for hierarchical actions [Erol et al., 1994]. This representation allows recursive definitions of methods (i.e. ways of reducing tasks), so that instead of having a fixed reduction schema (i.e. an abstract task always reduces into a given set of tasks) the reduction schema changes depending on the current state of the planning problem. This method allows problems with recursive elements to be expressed in a more natural manner, and is provably more expressive than the standard STRIPS notation [Erol, 1995, Chapter 5].

The following is an example of the notation used by UMCP using the Blocks World operators presented in Appendix B[1]. The goal of clearing the top of the block `A` is expressed as `clear(A)` (in UMCP this represents both an abstract task to make `A` clear, and the atom to be eventually added to the world state). This task can be reduced into either `DO_NOTHING` (in the case that `A` is already clear and therefore no action is necessary[2]) or `clear(?x)` $\rightarrow$ `unstack(?x,A)` (in the case where some block `?x` is stacked on `A` and must be made clear before it can be unstacked from `A` onto the table). The recursive nature of the second possible reduction enables a variety of situations to be easily expressed in HTN notation (e.g. the task of clearing multiple blocks stacked on `A`).

5.2 A Quality Measure for Abstract Plans

A quality measure is an integral part of an anytime algorithm. In [Zilberstein, 1996] it is stated that quality of partial solutions produced by an anytime algorithm should be both *measurable* and *recognisable*. The principle of measurable quality requires that the quality of an approximate result (i.e. not a full solution) can be determined accurately. The principle of recognisable quality requires that quality can be calculated at runtime without too much processing being required (e.g. in linear time). A quality measure is also necessary to determine whether the anytime algorithm satisfies a number of the other requirements of an anytime algorithm presented in [Zilberstein, 1996]. The most critical of these is whether the results returned from the algorithm are improving monotonically with respect to time[3].

Determining the quality of incomplete plans has traditionally been a difficult proposition. Completed plans are often compared on resource usage or length, but this is not particularly meaningful in domains that do not involve consumable resources or require plan length to be limited. In particular, there appears to be a dearth of domain-independent quality measures for HTN planners. In HTN planning, critics can be used to heuristically determine when a particular reduction should be pruned from the search space [Erol et al., 1995]. Critics are used to inspect partial HTN solutions to identify potential dead-ends and to resolve potential conflicts when they are first entered into a plan (e.g. when a resource is deleted by one action, but is required as a precondition for another action). Critics can also contain domain-specific knowledge and be used to remove particular unwanted situations from the planning process (e.g. when a particular set of actions should not be included in a plan together because it violates the behaviour of the domain). Although such critics are useful, they are only applicable in specific situations (e.g. when they spot the violation they are written to look out for), and are therefore not suitable for judging the quality of plans at every stage of the HTN planning process.

[1]This planning domain is based on the example operators that are provided with the implementation of UMCP.

[2]Do nothing actions act as terminating conditions that prevent infinite recursion. As such, all recursively defined methods must have a do nothing reduction.

[3]Determining whether plan quality is increasing could also be done without resorting to an explicit quality measure as long as a comparison operator existed that could determine which of two plans is “better”.

The quality measure that has been developed for A-UMCP is based upon the knowledge that an agent will not be able to directly execute the plans that it generates. Instead, any agent that uses A-UMCP will have to interpret any abstract plan that is produced after an interruption to the planning process. Although what this interpretation process actually accomplishes and how it is done can potentially vary from agent to agent, the general behaviour of such a process will be the same. The process must transform an arbitrary combination of abstract and primitive actions into some behaviours that achieve the goal that was planned for. However this is accomplished, the one constant fact is that a more abstract plan will require more interpretation to "fill in" the missing details than a less abstract plan. Also, this process of interpretation must happen extremely quickly, because the agent will interrupt the planner for a solution only if it requires the solution immediately (otherwise, given that the solution quality can only improve, it would be prudent to leave the planning process running). If an unrestricted amount of interpretation time was available, then interpretation could just be performed by an additional planning process, and hence have all the behaviours of planning associated with it (i.e. it would produce correct solutions). But, because the interpretation time is necessarily limited, the quality of any solution produced by any interpretation method will never be better than the result of using a planner on the same problem.

It is hard to define the behaviour of such an interpretation method at this stage because it will be based on the output of an arbitrarily interruptible process and the current state of an agent architecture, so this discussion has to rely on intuition about such a combination of an anytime planner, interpreter and agent architecture (although in Chapter 8 these concepts are presented in concrete detail). A good measure of success for the interpretation process is how similar the behaviour resulting from the interrupted plan is to the behaviour that would have been generated if the planning process had been allowed to construct a complete plan. The previous assumptions that more abstract plans will require more interpretation, and that it will be difficult for any interpretation process to perfectly mimic the planning process, leads us to make the further assumption that it will be more likely that a more abstract plan will be interpreted less successfully than a less abstract one. This can be illustrated with an example using the example HTN reduction presented in Figure 3.2. If the agent planning to do its shopping was interrupted before it had reduced the `Go To Shop` task, it would have to guess how to perform this action, and this guessing may be an unreliable and inaccurate process. This may lead to the agent having a bad plan for getting to the shops. On the other hand, if an interruption occurred after this reduction, then the agent would not have to guess how to get to the shops, so the resulting plan would not be as potentially risky (as it would include less guesses). To summarise this, an agent is likely to gain more information about how it should act from a plan that is less abstract, than from one that is more abstract. More information will then lead on to behaviour that is better (e.g. makes less mistakes or is more efficient) than less informed behaviour. Therefore, the quality measure for the plans produced by A-UMCP is based on how abstract the approximate result (i.e. the non-solution arbitrarily abstract plan) is.

If the planner was an anytime algorithm based on the progressive processing framework (as discussed in Section 4.4.1) calculating how abstract a plan is would be a fairly simple proposition. This is because it would be possible just to count how many processing levels the current plan is above the solution level. This approach would also provide the required monotonic increase in plan quality with respect to

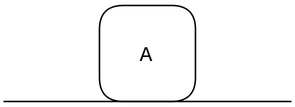

Figure 5.5: clear(A) requires 0 reductions.

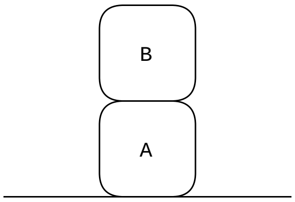

Figure 5.6: clear(A) requires 1 reduction.

processing time, as planning would only ever decrease the abstraction of the problem. Unfortunately the choice of UMCP means that we get a powerful recursive representation for HTN planning domains at the expense of not having convenient method of measuring how abstract UMCP plans are. This is because the UMCP notation does not provide a static and discrete notion of abstraction (as precondition removal planning does).

How the nature of the recursive notation affects the ability to simply measure abstraction is demonstrated in the following situations. The situations use the Blocks World notation which was introduced at the end of Section 5.1 and which is reproduced fully in Appendix B. In Figure 5.5, block `A` is clear so the abstract task `clear(A)` requires no reductions (except replacement with a `DO NOTHING` action). In Figure 5.6, block `A` is covered by one other block (`B`), so `clear(A)` requires one reduction into `clear(B)` $\rightarrow$ `unstack(B,A)`. Because `clear(B)` is already true (as `clear(A)` was in the first example), no further reductions are required. In Figure 5.7, `A` is covered by `B` and `C`, and working from our previous results `clear(A)` requires 2 reductions (1 reduction will make the situation similar to the previous example).

This shows that a single abstract task in UMCP can represent a different amount of action depending

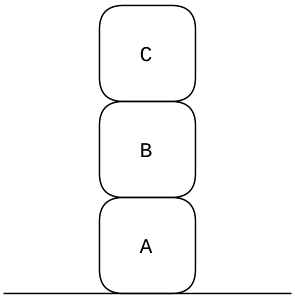

Figure 5.7: clear(A) requires 2 reductions.

on the current world state. `clear(A)` in the first example (Figure 5.5) is less abstract than the identical task in the second example (Figure 5.6), which in turn is less abstract than the same task in the final example (Figure 5.7). From this we can determine that any quality measure that intends to capture the notion of "abstractness" for UMCP plans must be recalculated any time the state changes (i.e. as the actions in the plan are changed) in order to accurately reflect the current state.

In order to calculate this abstractness-based quality measure for use in A-UMCP, it is necessary to view HTN planning from a slightly different perspective than usual. Instead of viewing abstract tasks as representations for groups of actions that must be performed by an agent, they can be viewed as representing a state-based planning problem to achieve a particular goal. Usually, each abstract task is present in an HTN plan to enable the agent to achieve a particular goal. This view is implicitly supported by the use of *goal tasks* in UMCP (i.e. the task `clear(A)` is used to represent achieving the goal `clear(A)`). This view is not supported by all HTN planners. For example, in SIPE [Wilkins, 1988], tasks are explicitly stated to represent units of behaviour, and not goal achievement. Although HTN planners manipulate abstract representation of actions, the only operators that can actually introduce literals into a state are the action primitives which behave in the same way as actions in a state-based planning problem. Given that abstract tasks can represent goals, and only primitive actions can be used to achieve goals, then a combination of primitives and goals gives us a state-based planning problem. For example, the previously discussed goal task `clear(A)` can be viewed as a state-based planning problem with the initial state being the initial state given to the HTN planner, and the goal state as the literal `clear(A)`. The state-based planning operators used to solve this problem are the primitives from the HTN planner (e.g. `dostack`, `unstack` etc. from the Blocks World).

Finally, we can relate this to how abstract the task is. As we have previously discussed, abstraction is a method for removing detail from the representation of problems. The more detail that is removed from the problem, the more abstract the representation is. When considering abstract HTN tasks as planning problems, the amount of detail removed is associated with the number of steps required to solve the planning problem represented by the task. A single task that represents a planning problem requiring a large number of steps to solve is more abstract than one which represents a planning problem that requires fewer steps to solve.

To calculate this measure of abstractness for use in A-UMCP, we can adapt the heuristic used by the Heuristic Search Planner (HSP) [Bonet and Geffner, 2001] to estimate the difficulty of solving a state-based planning problem to measure how abstract an HTN task is [Hawes, 2001]. HSP considers a relaxed planning problem (one in which the delete lists[4] from the STRIPS-style operators are ignored) to calculate "an estimate of the number of steps needed to go from state s to a state $s\prime$ that satisfies goal G" [Bonet et al., 1997]. This measure can be appropriated for calculating the abstractness of a task that attempts to introduce goal G. The HSP heuristic calculates this using the following formula (from [Bonet et al., 1997]), where p is an atom (e.g. a goal such as `clear(A)`), s is a state, and $g(p, s)$ is the estimated cost of achieving p from s.

[4]See Section 3.1 for a description of delete lists.

$$g(p,s) = \begin{cases} 0 & \text{if } p \in s \\ i+1 & \text{if for some } C \rightarrow p,\ \sum_{r \in C} g(r,s) = i \\ \infty & \text{if } p \text{ is not reachable from } s \end{cases}$$

This is summarised as follows; if the atom p is in state s then the cost is 0 (i.e. no action is required), if p is reachable from s then the cost is one plus the cost of achieving the preconditions of action C which introduces p into the state. If p is unreachable, then the cost is infinity. We can now use the following to calculate $g_{goal}(t)$ which is the cost of an abstract task t in an HTN plan from UMCP;

$$g_{goal}(t) = g(goal, S)$$

In this equation, the variable $goal$ is the goal represented by task t (e.g. `clear(A)` from the task `clear(A)`) and S is the current planning state. The current planning state is the initial state from the planning problem plus any extra atoms that have been added to it by previous actions in the plan. The cost $g_{network}$ of a whole task network tn can then be calculated using the equation;

$$g_{network}(tn) = \sum_{t \in tn} g_{goal}(t)$$

This equation ignores the presence of primitives in tn because they are not abstract, and therefore do not affect the cost of the task network.

This equation represents the ideal situation in which all tasks in an HTN plan are either goal tasks (abstract tasks representing a goal), or primitives. In reality, it is not always convenient to describe all abstract actions as goal tasks, and so UMCP allows for *action tasks* which are collections of other tasks (either goal or action tasks) and primitives, without one particular goal. Rather than artificially ascribe a goal to action tasks to use in a state-based planning problem, a separate algorithm is used to calculate their heuristic cost. The HSP heuristic basically calculates the minimum number of action primitives that must be applied to achieve a goal. This evaluation must be mimicked for action tasks. To do this, we can count the number of primitives present in the smallest possible reduction (i.e. one that represents the smallest amount of abstraction) of an action task. This is calculated by the equation;

$$g_{action}(t) = \sum_{pr \in t} 1 + min \sum_{a \in t} g_{action}(a)$$

Where pr represents a primitive action in t and a represents an abstract task in t. This calculation adds one to the cost of an action task for every primitive in the reduction. Finally we can then represent the cost of a task network tn as follows;

$$g_{network}(tn) = \sum_{t_{goal} \in tn} g_{goal}(t_{goal}) + \sum_{t_{action} \in tn} g_{action}(t_{action})$$

The cost of a task network is the sum of the costs of its goal tasks and its action tasks. This now gives us an approach to calculating how abstract a task network is and we can use this as the quality measure for partial plans created by A-UMCP. Appendix C presents an example of using this heuristic to calculate the costs of particular atoms for a given state in the Blocks World using the algorithm presented in the next section.

Calculating the abstractness of a task network by summing the cost of its individual tasks presents a problem because this approach can overestimate the cost of a task network. Calculating the costs of tasks separately (by treating them as goals to be achieved individually) ignores the possibility that tasks in a task network can interact and potentially "share" actions. If an abstract task earlier in the task network establishes a precondition of an action in a later abstract task, then it will no longer be necessary to have this action as part of the later task (this is one of the strengths of the UMCP notation). Considering tasks independently means that in this case the action will be considered to be part of both tasks, rather than just the first one. This leads to the estimated cost of the complete task network being larger than its actual cost (because the costs of overlapping actions are duplicated). This situation is resolved as the overlapping tasks are reduced and have their component actions added to the task network.

5.3 Algorithms for the Heuristic

To fulfil the requirement of an anytime algorithm having recognisable quality, the quality measure should ideally be calculated in at most linear time [Zilberstein, 1996]. In [Bonet and Geffner, 2001] the authors present an algorithm for calculating the above heuristic that has polynomial complexity in both the number of atoms and the number of actions. Although the complexity of this approach is worse than the desired case of linear complexity, the implementation of the adapted heuristic in A-UMCP is based on this algorithm. The algorithm is a forward chaining procedure in which the costs for all the atoms are initialised to zero if they are already in the state, or else infinity. Then, all applicable planning operators are applied and the results are added to the state (ignoring their delete lists). The costs for each atom are then updated to the minimum of either their current cost or the cost of achieving the preconditions of the action that add the atom into the state. This process is demonstrated in Appendix C.

Although this is a relatively simple procedure, due to the frequency with which it must be calculated it contributes a not-insignificant amount to the total running time of the anytime planning algorithm. Because of this, the basic algorithm for calculating the heuristic has been extended in two ways to increase its speed. The first extension is the relatively simple step of memoization. Because two states will always produce the same mapping of atoms to goals, it is safe to store previously calculated heuristic cost mappings and then index them by the state they were calculated from. If this state is then encountered again, the associated cost mapping is returned instead of processing it from scratch. Such situations tend to occur frequently within a single planning process as UMCP often produces task networks with identical task lists (and therefore states) yet with different constraints over the tasks (constraints do not affect the state used to generate the heuristic).

The second extension made to the heuristic calculation algorithm to improve its speed is more complex. The extension is based on the observation that because the delete lists of operators are ignored, the only way a state can be modified is through the addition of new atoms. Atoms already present in the

1. $i = 0$
2. Determine the set $diff$ which contains the atoms which differ between the current state S_{Ci} and the immediate ancestor state S_{Ai}.
3. Create a list op_i of the primitive operators that depend on one or more of the atoms from $diff$.
4. Generate $costlist_i$ from S_{Ci}
5. While $costlist_i \neq costlist_{i-1}$ do
   (a) $i = i + 1$
   (b) Apply each operator from op_i to S_{Ci-1} to generate S_{TEMPi}
   (c) Create list op_{i+1} of operators that depend on one or more of the atoms from S_{TEMPi}
   (d) Create S_{Ci} from the union of S_{TEMPi} and S_{Ai}
   (e) Generate $costlist_i$ from S_{Ci}
   (f) Store S_{Ci}

Figure 5.8: The extended HSP heuristic algorithm.

state cannot be removed or modified in any way. A consequence of this is that a state which is a subset of another set will have a cost mapping that is an ancestor (in terms of adjacent processing instances the heuristic) of the cost mapping of the other state. This fact can be exploited by calculating the cost mapping of a state from the cost mapping of its ancestor state (the state associated with the task network of which the current task is an immediate successor), instead of calculating it from scratch. This approach is valid because, using the above algorithm, the estimated costs can only reduce, and therefore the costs associated with a task network will either be the same as the ones associated with its ancestor (in the case that no cost-affecting changes have been made to the state) or lower (in the case that a cost-affecting change has been made, i.e. a new atom has been added to the state).

Before the algorithm for this extension is presented in detail, a high-level description of the process will be given. The costs for goals in a planning problem are calculated from the state generated by the current iteration of the heuristic algorithm. In the original version of the heuristic, atoms are added to the state in cycles as all possible operators are applied (none are deleted due to the delete lists of operators being ignored). In the extended version of the heuristic, rather than attempting to apply all of the planning operators, the extended version first adds to the state all of the atoms that were added in the equivalent cycle in the preceding heuristic process. It then only attempts to apply operators with preconditions that can possibly unify with any atoms that differ between the state just generated, and the equivalent state in the previous heuristic process. These states will only differ if the first heuristic state in the process (i.e. the one generated directly from the primitives in the plan) differs from the first state in the previous process. It then repeats this procedure for the next cycle of the heuristic.

Figure 5.8 presents a more detailed description of this algorithm. In this description S_{Ci} represents the state generated from the current task network after i cycles of the algorithm. S_{Ai} represents the same fact but for the immediate ancestor task network (the one being used as the basis of the calculation). The notion of dependency used in steps 3 and 5c is whether there is any chance that the precondition of an operator can unify with a given literal. If it can, then that operator depends on the given literal. Step 5f is needed to store the state that has been generated at cycle i so that it can be recalled by a later instance of the algorithm at step 5d as S_{Ai}. By introducing the atoms from the ancestor state cycle by cycle, the algorithm allows for changes in the state to occur due to the new additions in the plan. Even if atoms are later added from the ancestor state which could alter the values generated by the new state, the previously generated cost values will be lower and hence will be preserved due to the costs being the minimum of the previous and current values.

This extension speeds up the process because instead of attempting to apply every operator when generating the new state, only those that depend on those atoms newly added to the state are applied (the set of applicable operators is generated in step 5c). The process of applying an operator to a state is generally complex (each precondition of the operator must be tested for unification against each atom in the state) and minimising the number of times this action needs to be performed can lead to a large reduction in processing time. Empirical analysis of the efficacy of this extension to the HSP heuristic is presented in Section 6.4.

5.4 The A-UMCP Search Framework

To complete A-UMCP, the HSP heuristic method has to be associated with a search framework to explore plan space. A-UMCP is based around a greedy depth-first search. The greedy search framework maintains an open list of nodes that have yet to be expanded. Each node in the open list has its quality calculated using the heuristic described above, and then the list is sorted by these values, placing plans of highest quality at the head of the list. If two task networks have the same quality value, then they are sorted by the order in which they were added to the list, with newer task networks placed nearer the front of the list. This keeps the planning process expanding promising new branches without exploring older task networks of identical quality (further reasons for this are presented in Section 6.2).

As well as determining which task network should be expanded next, A-UMCP also determines which abstract task from the selected task network should be reduced next (UMCP allows any task to be selected). There are two interesting approaches to this. First, the planner could reduce the task with the greatest estimated cost first. This would be done to reduce the cost of the task network as quickly as possible (as reducing an abstract task should reduce its abstractness cost). The second approach is to always select the abstract task ordered earliest in the task network (cf. the left-wedge strategy used by ABTWEAK [Yang et al., 1991]). This would be done to produce task networks with more concrete actions earlier on. The advantage of this is that an agent using plans produced by A-UMCP will always have a plan with a more accurate beginning and it will therefore have something to base its initial actions on (perhaps giving it a chance to better determine actions later in the plan). Because this second approach is more agent-oriented, it is used within A-UMCP for selecting which abstract task to reduce next.

1. Get root node R
2. Set stored solution S_S = R
3. Set the open list O = $\{R\}$
4. While $O \neq \{\}$ do
    (a) S_C = $head(O)$
    (b) If $g(S_C) \leq g(S_S)$ and S_C contains no variables then $S_S = S_C$
    (c) Get set C, the successors of S_C
    (d) For $c \in C$ calculate $g(c)$ and insert into O
    (e) Sort O by g

Figure 5.9: The A-UMCP search algorithm.

The heuristic described previously will provide a plan quality measure that generally tends to increase as search progresses (or a cost measure that generally tends to decrease). This is because as A-UMCP plans develop they tend to reduce in how abstract they are (as this is the purpose of HTN planning). Unfortunately this does not provide the necessary guarantee of a monotonic increase in plan quality as processing time increases. To guarantee a monotonic increase in plan quality, the current best (i.e. the least abstract) plan is stored and is then available to be returned if an interruption occurs. This storage must be restricted slightly to prevent plans that contain uninstantiated variables from being retained. Allowing plans containing uninstantiated variables to be returned from A-UMCP would greatly add to the complexity of interpreting approximate solutions, because some actions would be, for practical purposes, unspecified. This is not a huge restriction because the A-UMCP search process usually assigns variables for an abstract plan before its constraints are updated.

The algorithm for A-UMCP is shown in Figure 5.9. In this algorithm S_S is the solution that is returned if an interruption occurs at any point during the planning process. Appendix D presents an example of using A-UMCP (although it is never interrupted) to find a solution for a planning problem in the Blocks World. This example also demonstrates how the costs of task networks reduce as the planning process proceeds.

5.5 Summary

This chapter introduced one of the principal contributions made by this thesis, the Anytime Universal Method Composition Planner (A-UMCP), an anytime planner based on the Universal Method Composition Planner (UMCP) [Erol, 1995], a sound and complete hierarchical task network (HTN) planner. The chapter started with a discussion of how a planner can be designed to truly fit the requirements of being an anytime algorithm. This included offering reasons why previous attempts at this had not fully demonstrated the necessary features of a true anytime algorithm. This was followed by a detailed discussion of the design of A-UMCP. Hierarchical task network planning was chosen as the basis for the anytime planner because it always has a representation of the complete plan available in case an interruption occurs. To enable an anytime algorithm to be constructed from UMCP, a quality measure for approximate solutions was presented. The quality measure calculates how abstract an approximate solution constructed by UMCP is. It does this in a manner adapted from the heuristic from the state-based planner HSP [Bonet and Geffner, 2001]. To speed up the calculation of the heuristic, a novel adaption of the HSP heuristic was presented that allows cost values to be computed incrementally from values stored from previous calculations. Finally, the A-UMCP search framework was presented. The framework is based on a depth-first biased greedy search and stores the current best approximate solution to guarantee that the results returned by A-UMCP improve monotonically with respect to processing time.

Chapter 6

Analysis of A-UMCP

This chapter presents an analysis of the Anytime Universal Method Composition Planner. This analysis covers the behaviour of the heuristic used to calculate the quality measure for A-UMCP, including its accuracy and how and why the speed of the calculation can vary. The general-purpose applicability of the heuristic is also considered, along with a comparison between the behaviour of A-UMCP and other existing anytime planners.

6.1 Applicability of the Heuristic

This section will examine how well the previously described heuristic encodes the notion of abstractness of task networks.

To evaluate the efficacy of the heuristic we must examine whether it adequately captures the concept it is being used to represent. We need to examine whether the heuristic used in A-UMCP adequately captures the notion of "abstractness". To do this we need to compare the values produced by the heuristic and some notion of "actual" or "measured" abstractness.

Before doing this, we need to be clear about what is being measured, and why. As stated previously, an "abstracted" plan representation is one that has had detail removed to leave a certain amount of information behind. To get an executable plan from an abstract plan representation, an amount of processing work must be done to fill in the remaining detail. The more abstract the representation, the more detail that is missing, and therefore the more processing time that is required to produce a plan. It is a claim of this thesis that the heuristic that is integral to A-UMCP can be used to estimate how abstract a task network is. To measure how accurate the heuristic is, a comparison must be made between how much time is taken by the planner to produce a solution from an abstract plan representation, and how abstract the heuristic estimates the representation to be.

The following tables show this comparison made for two domains. The comparison is made by recording how long it takes to produce a plan from each task network that is generated within a search process. The comparisons are made within a single planning problem because the variations in complexity between domains and problems mean that the amounts of time taken with different domains, or with

Estimate	**Time (s)**	σ
7	2.851	0.023
6	2.794	0.021
4	2.781	0.034
3	2.571	0.028
2	2.564	0.027
1	2.163	0.286
0	0.350	0.132

Table 6.1: Comparison between heuristic abstract estimate and average actual reduction cost for a Blocks World problem.

Estimate	**Time (s)**	σ
5	0.381	0.058
4	0.345	0.067
3	0.288	0.049
2	0.244	0.057
1	0.046	0.015
0	0.010	0.010

Table 6.2: Comparison between heuristic abstract estimate and average actual reduction cost for a Capture The Flag problem.

different initial states are incomparable. The results shown here are samples selected from a wider set of similar results. Table 6.1 is taken from the Blocks World. The figures shows that on average the heuristic correctly predicts how abstract a task network is. Table 6.1 contains no entry for a heuristic estimate of five, because no task networks of this amount of abstraction were produced in that problem. Table 6.2 supports these findings for the domain of Capture The Flag. The Capture The Flag planning domain has been developed specifically for this research, and it is the planning domain used by the agent using A-UMCP in an implementation of the Capture The Flag scenario. The planning domain can be found in Appendix A. The data used to generate these graphs were taken from a single run of the planner on a typical problem from each of the previously mentioned domains. The planning problems tackled in the examples are listed in Appendix E.

These results demonstrate that the heuristic can successfully discriminate between task networks of differing abstraction. The discrimination is not always perfect though, as in a small number of cases the boundaries of two heuristic values overlap, but it suffices for the majority of situations the planner tackles. There is no predictable relationship between the heuristic estimates and the actual time taken, but this was never intended to be the case. The results also demonstrate how the actual cost of an estimate varies over different planning domains. This can be seen in the large differences between the average time taken to reduce tasks with identical heuristic estimates.

6.2 General Behaviour

After verifying that the heuristic correctly estimates the abstractness of task networks, we can look at the effect using the heuristic has on the search process.

The way the heuristic is used in the search process is slightly different from the way heuristics are used in other search problems (e.g. Manhattan distance [Russell and Norvig, 1995, Chapter 4]). Whereas other search heuristics are used to guide the search process towards a solution in (hopefully) the most efficient manner, the A-UMCP heuristic is concerned with minimising a particular feature of the currently available solution to the problem. This distinction is an important one because the value being estimated by the heuristic (how abstract the task network is) has no direct bearing on whether the search node being evaluated is close to a solution or not. This is confused slightly by the fact that all solutions to HTN planning problems will have an abstractness estimate of zero (as they will be completely primitive), and therefore a less abstract task network could be closer to a solution than a more abstract one. Even though this is true, the heuristic provides no information about whether the task network has constraints that may potentially be violated, or abstract tasks without possible reductions. Solutions will certainly occur at the end of a downward abstractness gradient, but following such a gradient greedily will not necessarily lead to a solution.

What this means for search control within A-UMCP is that there is no informed method to lead it reliably to a solution. The constraint resolution methods from the original UMCP planner are able to prune non-solutions when they appear, but not predict in advance which reductions will lead to dead ends in the search space.

Although the heuristic provides no information about whether a task network is close to being a solution or not, it is used in combination with a depth-first search to control which task networks are presented as the next branch of the search tree to expand (i.e. the head of the open list). During search, the open list is first sorted by heuristic cost estimates. Then, if groups of task networks have identical costs, these groups are sorted by the order in which they were expanded, with newly expanded nodes being placed closer to the front of the open list than older ones (as they would be in depth-first search). Sorting by heuristic cost allows the search to reduce the cost of the available solution as quickly as possible. The depth-first ordering also encourages this because newly expanded task networks are more likely to lead to areas of lower abstractness than older task networks of identical quality. This is because reductions in abstractness tend to occur only after a task network has been the focus of a couple of search cycles, and so concentrating on the task networks that have recently been expanded produces quicker reductions in abstractness than distributing the search cycles over all the available expanded nodes.

This is backed up by experimental evidence comparing the real-time performance of a breadth-first version of A-UMCP to the real-time performance of a depth-first version on a variety of problems. Figure 6.1 shows a comparison between depth first and breadth first search for an example problem in the Blocks World. Figures 6.2 and 6.3 show the same comparison for the Capture The Flag and UM Translog domains respectively. UM Translog is a complex logistics-based planning system designed to benchmark modern planning systems [Andrews et al., 1995]. The problems used to generate these

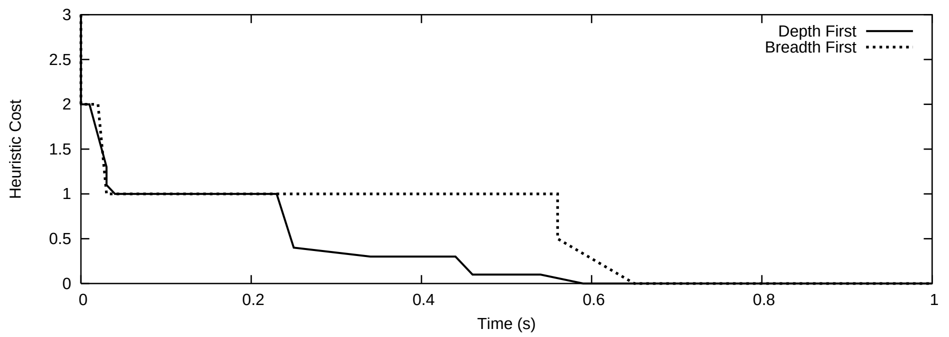

Figure 6.1: An example of depth and breath first search styles for the Blocks World Domain.

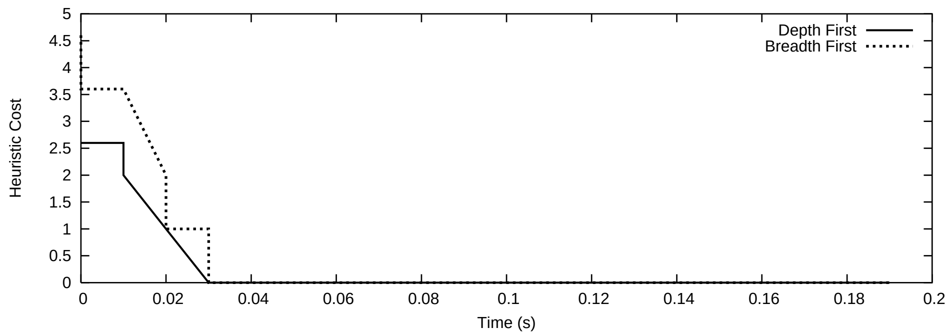

Figure 6.2: An example of depth and breath first search styles for the Capture The Flag Domain.

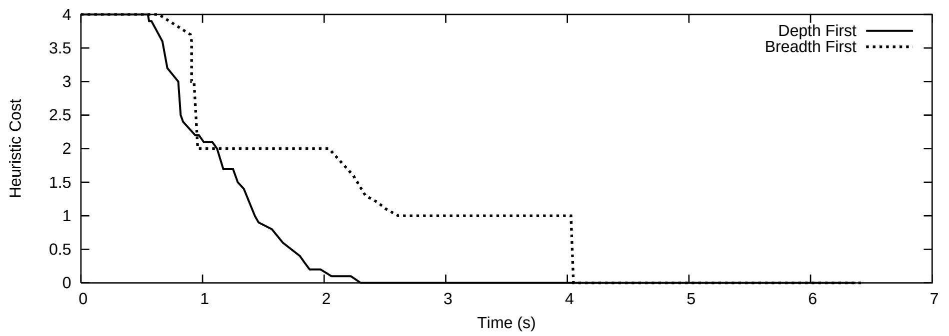

Figure 6.3: An example of depth and breath first search styles for the UM Translog Domain.

graphs can be seen in Appendix E.

6.3 Performance Analysis

To allow an anytime algorithm to run successfully in real-time, the cost measure it uses should not impact too greatly on the running time of the algorithm. In the anytime algorithm literature, it has been suggested that the cost measure should have at most linear time complexity with respect to problem size. In the literature describing the original version of the A-UMCP heuristic it is stated that the implementation of the heuristic that has been adapted for use as the basis of the A-UMCP heuristic runs in polynomial time [Bonet and Geffner, 2001]. Unfortunately this is greater than the suggested linear time. Although the time complexity of the heuristic algorithm remains unaltered, the actual running time of the algorithm has been reduced in comparison to the original algorithm by the addition of the extensions that were presented in Section 5.3 (see the next section for details on their effects).

In the implementation of A-UMCP, the amount by which the processing of the original algorithm is slowed by the heuristic calculations greatly depends on the domain and the problem being tackled. The speed of the heuristic calculations greatly depends on the number of atoms in the search process. This dependency is due to the fact that an increased number of atoms lead to larger increases in state size during the operator application phase of heuristic calculation. Larger state sizes then lead on to require an increased number of operator applications, further degrading the real-time performance of the algorithm.

The heuristic makes the least impact on the running time of the planner when it can utilise the memoization and incremental compilation extensions. These extensions are used more often when a greater number of search nodes are being expanded and hence more consecutive calculations are performed (i.e. when a problem is harder to solve). The heuristic makes the greatest impact on the running time of problems in which only a few task networks are expanded before a solution is found. These extremes are demonstrated in Figures 6.4 and 6.5. As before, the graphs represent the results of the planner solving a single problem in one of the example domains. The problems used to generate these graphs are listed in Appendix E. On the graphs, the x-axis represents planning time and the y-axis represents plan quality. The experiments end when the first solution is found to the planning problem. The results are from individual planning processes using both a pure depth-first search strategy and the A-UMCP search strategy. Figure 6.4 shows an example where the heuristic version actually outperforms the non-heuristic (depth-first) version by producing task networks of a lower cost faster in real-time. Figure 6.5 shows an example where the non-heuristic version dramatically out-performs the heuristic version when it is faced with an example of the extreme situation described above. It is important to note that the heuristic version is never out-performed by the non-heuristic version when time is measured in terms of search cycles (i.e. not in real-time, so that the processing time of the heuristic is ignored).

Away from these two extremes, the heuristic does alter the time taken by the original planning algorithm, but never to such great extents. On average, the amount of time to find a completely primitive plan in the heuristic version is approximately twenty-five percent longer than the time taken by the non-heuristic version. It is also very rare that the heuristic version decreases plan cost faster than the

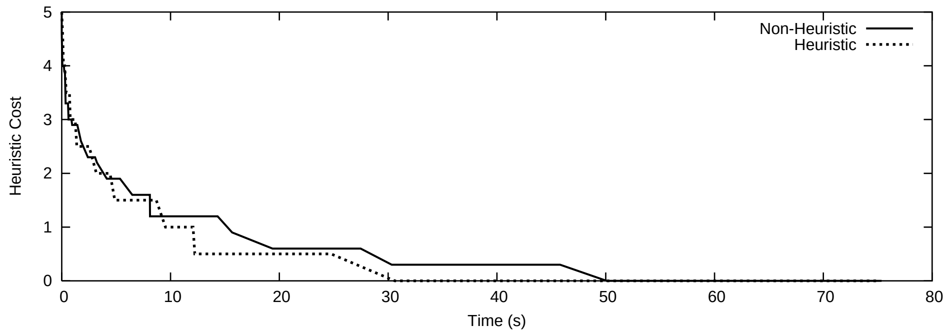

Figure 6.4: Heuristic and Non-heuristic versions of A-UMCP in the Blocks World. In this example the heuristic version outperforms the non-heuristic version in real-time.

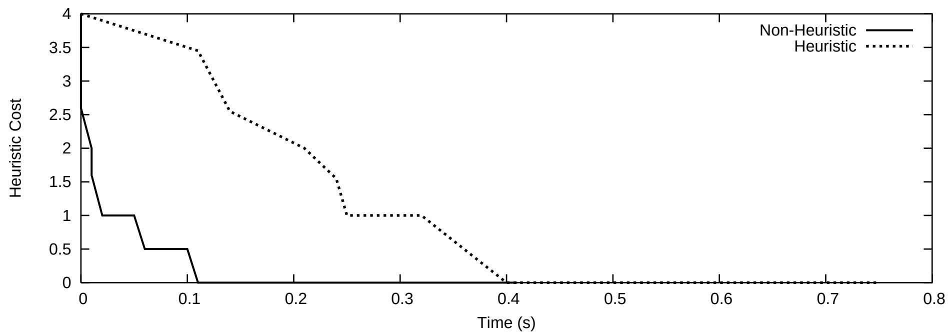

Figure 6.5: Heuristic and Non-heuristic versions of A-UMCP in the Blocks World. This example shows the heuristic version being dramatically outperformed in real-time by the non-heuristic version.

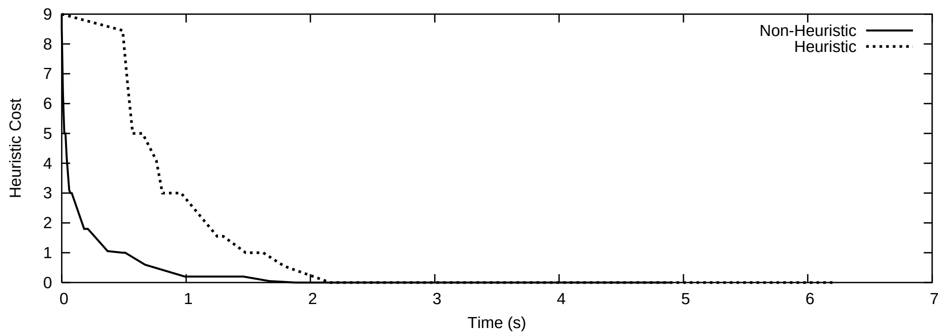

Figure 6.6: Heuristic and Non-heuristic versions of A-UMCP in the Blocks World.

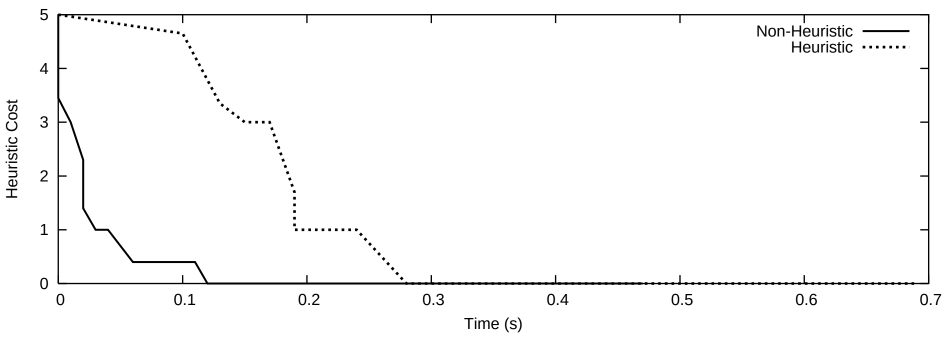

Figure 6.7: Heuristic and Non-heuristic versions of A-UMCP in the Blocks World.

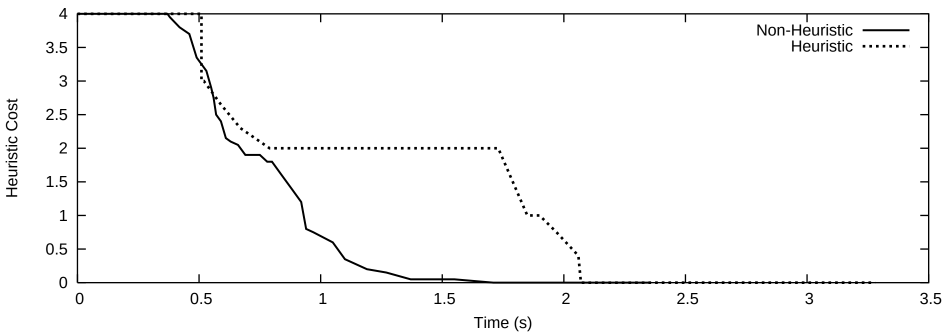

Figure 6.8: Heuristic and Non-heuristic versions of A-UMCP in the UM Translog domain.

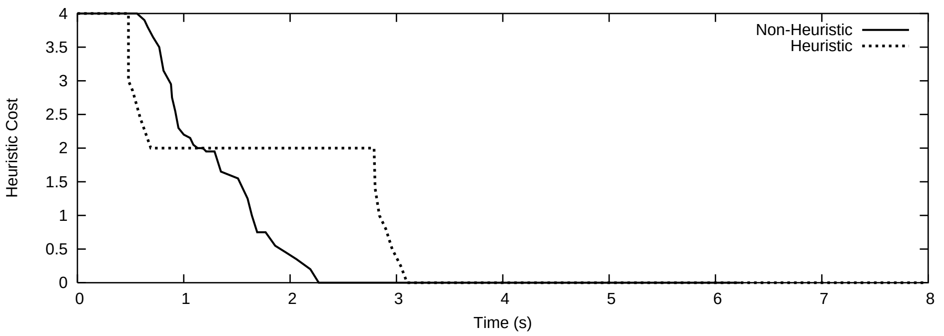

Figure 6.9: Heuristic and Non-heuristic versions of A-UMCP in the UM Translog domain.

non-heuristic version in real-time. The following examples demonstrate the behaviour of the heuristic and non-heuristic versions of A-UMCP on a selection of problems and domains. Figures 6.6 and 6.7 show comparisons for two problems in the Blocks World. Figures 6.8 and 6.9 show comparisons for two problems in the complex UM Translog domain. The results demonstrate that the version of A-UMCP using the heuristic is generally slower than the version using pure depth first search. Occasionally portions of the graphs show the heuristic version increasing the quality of currently available solution faster than the uninformed version (e.g. the start of Figure 6.9). This comes from the sorting of the open list by quality. Unfortunately these increases are usually outweighed by the time required to calculate the heuristic values. Obviously it is never desirable to slow down a planning process, particularly in a real-time domain, but the heuristic is an integral part of A-UMCP and as such must be included in the process. The reduction in planning speed is an unfortunate but necessary side effect of creating an anytime planner.

6.4 Effectiveness of the Algorithm Extensions

The real-time performance of A-UMCP has been significantly improved by the extensions made to the algorithm used to calculate the heuristic. These extensions were presented in Section 5.3, but to re-cap they are the memoization of calculation results, and the incremental calculation of the heuristic.

Of the two extensions to the algorithm, the one with the smallest effect on the speed of heuristic calculations is the incremental calculation extension. The degree to which this extension affects the speed of the algorithm is dependent on how much of the current calculation has been performed previously. If very little has been previously calculated, then its effect will be minimal. If a greater amount has been calculated, then its effect will be greater. Because of this dependency, planning processes that run for a greater number of cycles (particularly those in which these cycles tend to work on nodes from a common root) will garner a greater benefit from the extension to the algorithm. When the extension is used in planning processes that do not run for many cycles, the overhead associated with the extension will sometimes render the algorithm slower than the unextended version. The incremental calculation extension also has a greater effect when the heuristic is more complex to calculate (e.g. a greater number of interacting objects in the problem). This is demonstrated by the differences between Figure 6.10 and Figure 6.11. They show data taken from the planner being run on identical problems with different initial states. Figure 6.11 has a much bigger initial state (in Figure 6.10 the non-critical objects have been removed), and therefore the heuristic takes longer to calculate. This is reflected by the greater effect made by the incremental extension in Figure 6.11. In Figure 6.10 the extension can be seen to have a detrimental affect on processing time, whereas in Figure 6.11 the effect is positive (notably when no memoization is used). The problems used to generate these graphs can be seen in Appendix E.

Because of this, it would be prudent to only apply the incremental calculation extension when solving problems that take above a certain number of cycles to solve, or have a particularly complex heuristic calculation (where pruning of the state is not possible).

The memoization of heuristic results provides a much greater decrease in processing time. The extent

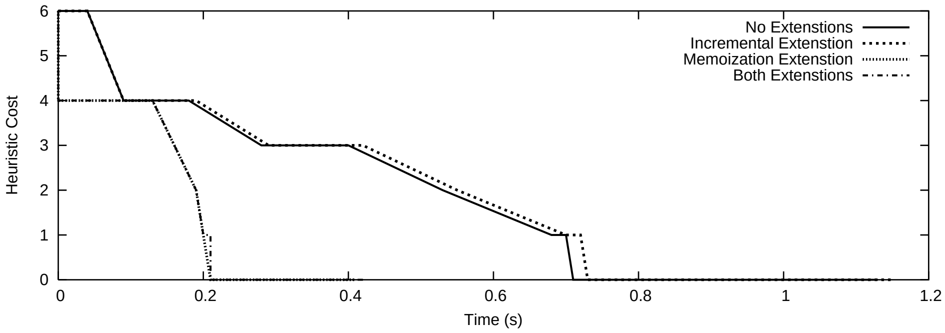

Figure 6.10: The effect of the algorithm extensions on the processing speed of A-UMCP for a Capture The Flag example problem.

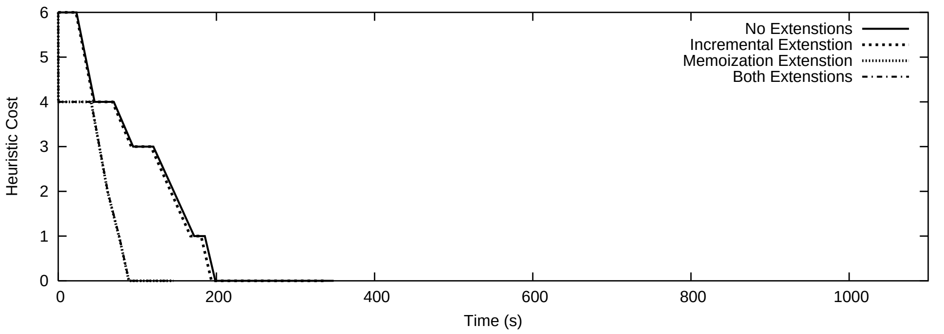

Figure 6.11: The effect of the algorithm extensions on the processing speed of A-UMCP for a Capture The Flag example problem with an expanded initial state.

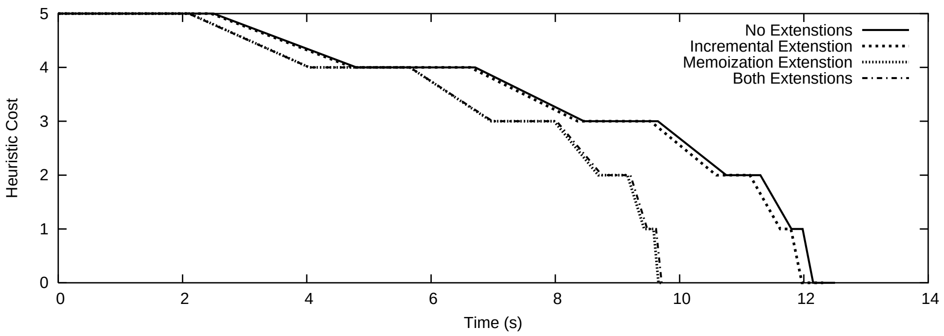

Figure 6.12: The effect of the algorithm extensions on the processing speed of A-UMCP for a Blocks World example problem.

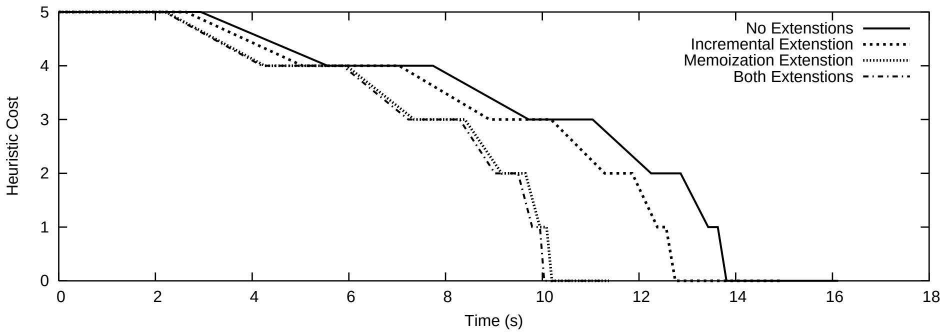

Figure 6.13: The effect of the algorithm extensions on the processing speed of A-UMCP for a Blocks World example problem.

of the effect is dependent upon how many duplicate states are encountered within the planning problem. In the heuristic's original application in state-based planning, the number of duplicate states encountered may be minimal because every new planning cycle will add new atoms to the current state through the addition of a new primitive to the plan. In HTN-based planning, memoization is more effective for three reasons. First, planning cycles do not always add new actions to the current plan, as processing cycles also perform ordering and constraint satisfaction steps, therefore HTN planning features more cycles with identical states. Second, new additions to plans do not always result in the current state being altered (e.g. `DO_NOTHING` actions and abstract tasks). Third, due to the complete nature of the original UMCP algorithm, refinements to task networks often produce a number of task networks that are identical apart from one or two constraints. The subsequent refinements of these almost identical task networks will result in numerous task networks that are almost identical. All of this similarity will produce many situations in which heuristic calculations can be entirely reused, and therefore result in the memoization extension having a greater effect.

When the two algorithm extensions are combined, their previously described dependencies are retained. This makes little difference for memoization as the conditions in which it improves performance are still present. The addition of memoization does have an impact on the effectiveness of incremental calculation. Before memoization is introduced the incremental calculation produces a saving in computation time for each duplicate calculation that is performed by the heuristic. When these duplicates are removed through memoization, the incremental calculation extension only produces a saving in computation time the first time a particular task network is encountered. This reduces the effectiveness of the incremental extension. When the extension is not effective it occasionally renders the calculation slower than the non-extended version. This behaviour can be seen in most of the results presented in this section, e.g. Figures 6.12, 6.13, 6.10, and 6.11.

From these results we can conclude that memoization of the results of the heuristic should always be performed as it has a large impact on the running time of the algorithm. The same is not true for the incremental calculation extension. These extensions to the HSP algorithm are general purpose, so any planner that uses the HSP heuristic can use them for a performance gain. But, due to the fact that the incremental calculation extension can sometimes slow down planning rather than speed it up, some analysis should be performed on the domain before this extension is applied. Given the nature of the incremental extension, it might be more suitable for application in state-based planning problems as they gradually build up a state from a common starting point.

6.5 The Heuristic as a Critic

Although the heuristic used in A-UMCP is domain independent, it has only limited possibilities for application outside of the A-UMCP framework. Within the A-UMCP framework the cost measure is used as a critic to remove invalid `DO_NOTHING` actions as they are introduced into the search space. Every recursively defined method must have a `DO_NOTHING` refinement in case it is already true in the current state (as described in Section 5.1). This means that every time a recursive method is refined, an additional, possibly invalid, task network containing the `DO_NOTHING` primitive is added into the search space. The A-UMCP heuristic allows dead-end task networks that are generated by refinements of recursive methods to be pruned from the search space before they are further examined by the planning process. This is done by examining the heuristic cost of the task that the `DO_NOTHING` is refined from. If the parent task of a `DO_NOTHING` reduction has a cost estimate of zero it means that the literal it represents is true in the current state. This means that the task network has a valid `DO_NOTHING` refinement which can be left in the search space. If the parent task network has a cost greater than zero, then the `DO_NOTHING` reduction is invalid and should therefore be removed from the search space.

This feature of the heuristic is useful, and it is completely general purpose. This means that it could be implemented in all planners based on the UMCP recursive notation. Unfortunately, the performance gain provided by pruning dead-end `DO_NOTHING` reductions is not large enough to warrant the inclusion of the A-UMCP heuristic, as its calculation adds considerably to the running time of A-UMCP.

6.6 Performance Affecting Domain Features

Some planning domains are especially suited to the application of the A-UMCP heuristic. In particular, domains that can be entirely represented in the recursive notation allowed by UMCP are most suited. This is because the method described in Section 5.2 for calculating g_{action}, the heuristic for non-recursive abstract tasks (i.e. action tasks), sometimes leads to the production of results that do not compare suitably to those produced for recursive tasks (i.e. goal tasks and action tasks representing similar amounts of abstraction are assigned different heuristic costs). By only storing the minimum cost for a particular method (a decision made to mirror the HSP algorithm), any task that could possibly evaluate to zero has its value set at zero. This is because the cost of action tasks are calculated before runtime, as unlike goal tasks they can have static costs. Statically calculating the cost of action tasks was done as an optimisation to improve the speed of the heuristic calculation. Unfortunately, action tasks can contain goal tasks, so their costs should really be calculated every planning cycle. This represents a flaw with the heuristic used in A-UMCP and is something that should be considered if any future work is done with A-UMCP.

Because of this weakness, the heuristic sometimes under-estimates the cost of task networks when non-recursive abstract tasks are involved. Under-estimation of heuristic cost also occurs when a situation arises that causes the relaxed assumption of the heuristic to be violated. The relaxed assumption causes the heuristic to ignore the delete lists of the planning operators and therefore assign a lower cost to a series of actions that would have a higher cost if the delete lists were considered. Such situations occur when one action clobbers (i.e. deletes) the precondition of an action later in a plan, and additional actions are needed to reinstate the precondition. The additional actions are ignored by the heuristic and hence the heuristic value is lower than it should be for the task network.

An underestimate (caused by either of these problems) can cause a task network to be stored as the current solution when it is actually more abstract than the previous solution. This could possibly cause problems if an agent was waiting for a plan of specific actual quality (rather than measured quality) before interrupting A-UMCP, but it is an unavoidable side effect of the heuristic (the second problem is present in its original form as well).

6.7 Comparison to Existing Anytime Planners

The planner that has been created has several key differences from the other anytime planners reviewed previously.

The most important of these is that A-UMCP supports "true" anytime planning. This means that the planner can be interrupted at *any* time, and not just once a particular point in the planning process has been passed.

Of all the anytime behaviour planners reviewed, A-UMCP is one of the few that provide true anytime behaviour. A-UMCP provides this behaviour by having no initialisation phase (a term introduced in Section 4.5), and therefore no delay before the first point in processing that allows interruptions. In A-UMCP, as soon as the planning process has been initiated, a solution is available if an interruption occurs. This solution is the root node of the HTN planning problem, which is then replaced by subsequently more refined (and less abstract) solutions. This type of behaviour treats the root node as an initial guess at a solution, and therefore A-UMCP shares the characteristics associated with other planners that exhibit such behaviour (see Section 4.5).

Allowing any solution (abstract or primitive) from an HTN planner provides its "true" anytime behaviour, but it also means that an additional plan interpretation step is required. This step is not explicitly mentioned with any other anytime planning algorithm except the one presented in [Briggs and Cook, 1999], although it is quite possible that other algorithms will require a step to translate between a planning representation and an acting one (possibly doing some ordering and instantiation at the same time). The notion of a solution for A-UMCP is altered from the traditional planning aim of producing a series of actions that achieve a goal, to producing a structure that provides as much information as possible to a subsequent interpreter component in an agent architecture. A complete primitive solution to an HTN planning problem provides the greatest possible amount of information to the interpreter component, whilst the root node of a planning problem provides the least. This approach to the notion of a solution groups A-UMCP with other planners that allow a "wider" class of solution (a notion introduced in Section 4.5). Within this group, it can be associated with the approaches that use abstract representations, rather than those that specify constraints over primitive solutions. As far as it can be determined from reviewing the available literature, A-UMCP is the only implemented action planner that uses this particular approach to the notion of what is a solution.

6.8 Summary

In this section we have analysed the performance of the Anytime Universal Method Composition Planner (A-UMCP), a planner which has been produced to overcome some of the weaknesses displayed by traditional planning when used in environments that are real-time, dynamic and complex. This section examined and analysed the real-time performance of A-UMCP, paying particular attention to the calculation of the heuristic used to determine how abstract a task network is. This focus was necessary because (as was demonstrated in Section 6.3) the calculation of the heuristic has a significant impact on the running time of the planner. Extensions to the basic heuristic algorithm designed to lessen this impact were evaluated, and it was found that the memoization of heuristic results greatly reduces running time, whilst incremental calculation of the heuristic only has positive effects given certain properties of the problem being tackled. The section ended with some qualitative analysis of the planner. This involved considering the general purpose applicability of the planner's heuristic, looking at the features of domains that work well with the planner, and finally comparing the planner to other similar anytime planners. A-UMCP is evaluated in context, as an integral part of an agent architecture, in Chapter 9.

Part III

A Hybrid Agent For Computer Game Worlds

Chapter 7

Review of Agent Literature

This chapter reviews work in the field of agent design related to the objectives of this thesis. In particular, this review concentrates on work relating to computer games and agents embodied in real-time, dynamic environments.

7.1 Reactive Agents

In previous chapters in this thesis, the need for agents in dynamic worlds to react quickly to stimuli has been emphasised frequently. This need for processing speed naturally leads to the consideration of processing methods that are equally rapid. In other domains that traditionally require fast methods of generating agent behaviour (e.g. robotics), processing is usually done in a purely reactive manner. As with reactive planning (see Section 3.2.2), reactive approaches to agent design aim to encode every possible result of a processing system (in this case, a behaviour generation system including sensing, processing and acting) in a series of rules. The contents of a knowledge base are matched against the antecedents of these rules. If a match is found then the consequent of the matched rule is executed. The principal behind reactive agent approaches is that the world is its own best representation, and any additional knowledge representation is superfluous. This stance was popularised by Rodney Brooks [Brooks, 1986, Brooks, 1991] and has been challenged in a variety of places (e.g. [Kirsh, 1991]). Examples of reactive agents and approaches to building them include the Agent Network Architecture [Maes, 1991] and work done by Leslie Pack Kaelbling on reactive systems [Kaelbling, 1990]. Reactive agent approaches especially intended for computer games or game-like worlds include work on the Pengi agent [Agre and Chapman, 1987] and more recently work by [Aylett et al., 1999], [Horswill and Zubeck, 1999], [Shapiro, 1999] and [Khoo et al., 2002]. The philosophical foundations of reactive behaviour generation will not be examined here. Instead, we will examine some selected benefits and restrictions associated with reactive agents and how these benefits and restrictions relate to the aims of this thesis.

Generating behaviour in a reactive manner has a number of benefits. The principal benefit is the high speed at which an agent can determine its next action. This is due to the low processing overheads (particularly when compared to something like planning) involved when matching the current situation to a behaviour rule. As already stated, this ability to quickly determine the next action to take is a desirable property for an agent in a computer-game world. Approaching agent implementation in a reactive manner also has the advantage that it makes agent behaviour responses easy to construct. Agent behaviour in particular situations can be specified without worrying about side effects produced through interactions with other behaviours (because only the matched behaviour will be executed), and idiosyncratic reactions can be produced in response to important (i.e well-being threatening) stimuli.

The disadvantages that are associated with solely reactive agents are similar to the disadvantages that afflict all purely reactive methods of behaviour generation. Importantly, these disadvantages also conflict with the aims of this thesis. As stated previously, the very nature of reactive behaviour generation is that all behaviour is based on the immediate situation. This precludes the agent from achieving situations in the future unless the behavioural paths to these situations have been accurately anticipated during the agent's design. Circumstances that have not been anticipated may produce unexpected or unwanted behaviour from the agent. This weakness prevents the agent from demonstrating anything but the simplest of goal directed behaviour (i.e. behaviour that will *always* achieve a goal when a particular stimulus is present). Another problem with reactive methods is that although in a simple domain (e.g. the Pengo world [Agre and Chapman, 1987]) the processing overhead of matching the antecedent of a reactive rule to a situation is small, this does not scale up well when faced with complex domains (e.g. a modern computer game). This is because matching a set of conditions against a set of stimuli suffers from a combinatorial explosion [Ginsberg, 1989]. Therefore, giving an agent an interesting set of behaviours in a complex world will result in an extensive processing task to retrieve the correct reactive behaviour. A similar problem also affects deliberative systems in complex worlds (some amount of complexity is unavoidable).

One attempt to make reactive systems more goal-directed is Nilsson's Teleo-reactive Program (TRP) formalism [Nilsson, 1994]. As with other similar approaches (most notably the PRS and RAP reactive planners reviewed in Section 3.2.2), the TRP system organises its reactive rules into units (known as Teleo-reactive Programs or TRPs) that achieve particular goals. What sets the TRP system apart from other reactive systems is the way rule conditions and effects are structured within the units. To enable each TRP to achieve its specified goal, production rules are ordered within it so that each action establishes the conditions necessary for another rule higher up (evaluated earlier) in the TRP to fire, bringing the agent one step closer to its goal. To provide the flexibility to deal with unexpected environmental events (i.e. those not relevant to currently executing action), the conditions of the TRP rules are continuously evaluated. If an earlier condition for rule execution is found to be satisfied, then this rule is executed. Actions for rules in TRPs can be other TRPs, thus allowing the agent designer to specify behaviour in a hierarchical manner. These abilities allow an agent based on the TRP formalism to achieve a better degree of goal-directness than if it was using other reactive paradigms. Unfortunately, a purely TRP-based agent still requires that the agent designer anticipates all of the important environmental events that can affect the agent. In addition to this, the constant evaluation of the rule conditions in all the active TRPs means that this method of reactively generating behaviour is also a potential victim of the indexing and scalability problems associated with other reactive systems.

Because we want to give an agent the ability to perform interesting, intelligent behaviour directed towards complex goals in complex environments, we have to look beyond purely reactive agents because even though they provide some of the necessary features (e.g. processing speed) they fail to provide others (e.g. reliable, complex goal-directed behaviour).

The issues that arise when investigating the trade-off between encoding a process deliberatively and encoding the same process reactively is discussed at a number of points in Chapter 8.

7.2 Hybrid Agents

Just as agents can be purely reactive, it is also possible to construct an agent that is purely deliberative. This is equivalent to taking a planner and adding processes to handle sensing and primitive execution. We have already seen the myriad problems faced by planners in complex, dynamic worlds (see Section 2.3.3), so, for the same reasons, this concept can be dismissed fairly easily. To develop an agent that displays the behaviour required by the aims of this thesis, we must look beyond a purely reactive or purely deliberative approach and investigate hybrid approaches to agent design. A hybrid approach involves combining the main strengths of a variety of methods (e.g. reactive rules for fast reactions and a deliberative planner for goal-directed behaviour) in an architecture that controls their execution.

There are many ways in which hybrid agents can be constructed, and hence numerous different examples in the literature. Before specific examples are reviewed, the CogAff architecture schema[1], a general framework for developing agents of variable composition, will be presented. The agent research presented in the rest of this thesis is guided by the principles of the CogAff schema, so the following section also provides an introduction to the concepts involved in building an intelligent agent. Because (as will be seen in the next section), the CogAff framework is quite general, the choice to situate this thesis within it provides very few restrictions whilst providing a wealth of previous research to build on. As such, it is hoped that the following overview of the CogAff schema also provides the necessary justifications for its use.

7.2.1 The CogAff architecture schema

In the most general terms, the CogAff architecture schema is a collection of concepts and relationships with which (and within which) an agent architecture can be described, designed, built and compared to other architectures. A high-level description of the CogAff schema and its purpose can be found in [Sloman and Scheutz, 2002]. In addition to this, numerous other publications have been dedicated to the CogAff schema, some of which will be presented here.

The CogAff architecture schema is presented in Figure 7.1. It is divided into three towers (the vertical divisions) and three layers (the horizontal divisions). These general divisions have been used by various

[1]The name "CogAff" comes from the name of the research group that developed the approach, the Cognition and Affect group at the University of Birmingham.

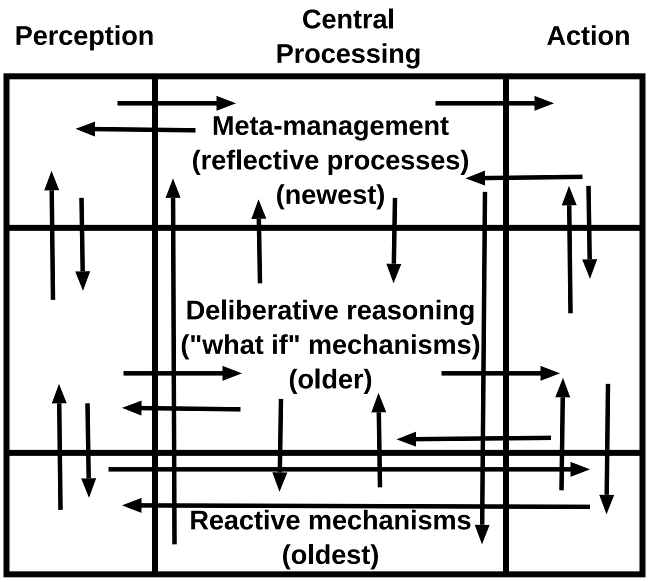

Figure 7.1: The CogAff Architecture Schema [Sloman and Scheutz, 2002].

other authors, for example [Nilsson, 1998, Chapter 25]. Most previous work on architectures agree on common names and purposes for the three towers. The left tower is dedicated to sensing the agent's environment. The middle tower is dedicated to the agent's internal processing. The right tower is dedicated to carrying out actions in the agent's environment. Where other architectures have featured three (or more) common layers, there has been a great deal less agreement about the purpose of each horizontal layer. Some more detailed examples will be presented in the following section, but often a layer is dedicated to a particular processing component (e.g. SIPE-2 in the Cypress architecture [Wilkins et al., 1995]). The CogAff architecture schema goes beyond this by only specifying the processing style for each layer (not a specific component). The bottom layer performs reactive processing. The middle layer performs deliberative processing (e.g. generating and reasoning about possible future states). The top layer performs meta-management (reflective) processing (e.g. monitoring and reasoning about internal processes). Each layer operates concurrently, with no dominance hierarchy. The reactive layer is typically used to handle situations where a fast response is required (e.g. to move the agent quickly out of danger), or where the agent needs to coordinate its actions with immediate feedback from its environment (e.g. tracking a moving object). The deliberative layer is used to perform tasks such as reasoning about the future states an agent may want to achieve (e.g. planning to achieve a goal state), or manipulating data that is necessary for this reasoning (e.g. maintaining a database of beliefs and motives). The meta-management layer is typically used to evaluate how the agent is performing and make any necessary changes to the way it processes information.

There have been a number of agents that have been designed and implemented within the CogAff schema. The work of Luc Beaudoin [Beaudoin, 1994] and Ian Wright [Wright, 1997] spans two theses and represents some of the earliest efforts developing and applying the ideas behind the CogAff schema.

The results of this work also demonstrate the strengths of the structure offered by the CogAff architecture schema. Beaudoin's NML1 architecture was designed to demonstrate advanced goal processing abilities. Wright's MINDER1 architecture and subsequent implementation builds on NML1 to produce an agent that is capable of demonstrating perturbant states. Both agents were designed to operate within the nursemaid scenario. This scenario places the agent as a nursemaid in charge of a nursery full of babies. The babies can fight, fall into ditches and do other things that will require the attention of the nursemaid. It is the nursemaid's job to attend to all of these situations with the correct behaviour (separating fighting babies, re-energising babies etc.). Although it wasn't intended at the time, this world has similar traits to those of a computer game world, namely dynamism and a relatively complex choice of goals and actions. This similarity means that any agents that are successful in this environment will have features that are desirable for some computer game agents. NML1 was never implemented, but MINDER1 was, and operated successfully. The success referred to is not based on the notion of success used in the original research (i.e. whether perturbant states were evident), but whether the agent "overcame" the real-time nature of the environment it was situated within.

The MINDER1 agent performed successfully in this dynamic environment for two reasons. The first reason was that it had very little deliberative processing to do, thus it was always free to react to environmental changes. The second feature that made it successful was the underlying agent architecture (based on the CogAff schema). The design of the architecture is such that an agent based on it can make the most of its limited processing resources, whilst also being able to react to important environmental events in a timely manner. Both these facts are crucial to the successful implementation of intelligent agents in computer games. The ability to use processing time effectively in MINDER1's deliberative and meta-management layers is supported by the attention filter. The filter is used to inhibit the progress of goals (requests for processing) generated by the reactive layer. Its basic operation can be summarised as thus; every goal generated is assigned a heuristic measure of insistence, if this level is greater than the one encoded by the filter, then it is allowed to "surface" and influence processing (i.e. be considered for adoption by the agent) in the deliberative layer [Sloman, 1993a]. In addition to this, a timely (although not always optimal) response to environmental change is supported by having concurrently operating layers. If the deliberative layer is engaged in a processing task, the reactive layer will always be available to react to the current situation.

Because the CogAff architecture schema does not enforce the presence of any of the features identified as strengths of the MINDER1 architecture, it would not be accurate to say that adopting the CogAff approach for an agent design guarantees the adoption of these advantages. The truth is that by examining how architectures function, and by comparing them to other architectures that are either based on the CogAff framework or that can be mapped onto the framework, we can gain a better understanding of what the relative benefits of certain approaches are, and why they provide these advantages. An example of such a comparison process can be seen in [Scheutz and Logan, 2001]. Investigations like this are much easier when working within the CogAff schema because everything can be discussed in terms of previously established concepts and using previously defined terminology. This avoids the conceptual confusion that is evident when researchers with different motivations and backgrounds discuss similar topics (examples of this are given in [Sloman and Scheutz, 2002]). A further example of research that compares architectures (outside of the CogAff framework) can be seen in [Bryson, 2001].

7.2.2 TouringMachines

Ferguson's TouringMachines [Ferguson, 1992] are hybrid agents that operate in a driving domain (the TouringWorld). Driving is an interesting domain because it combines long term goal-oriented behaviour (e.g, getting from A to B) with short term dynamic behaviour (e.g. avoiding parked cars and stopping at traffic lights). The ability to focus on long-term goals whilst dealing with the current situation demonstrates the type of behaviour we require from computer game agents. TouringMachines are based on an architecture similar to the nursemaid agent MINDER1, which is consequently within the CogAff schema. The TouringMachine architecture features a reactive layer, and a planning layer. The planning layer is similar in function to the CogAff's deliberative layer, but is used solely for planning. The planner used in the implementation of the TouringMachines is more sophisticated than MINDER1's deliberative layer. It performs hierarchical planning, and also interleaves planning and execution. The critical difference between the architectures of MINDER1 and a TouringMachine appears in the top layer. Where the CogAff architecture has its meta-management layer, the TouringMachines have a modelling layer. The modelling layer is used to construct behavioural models of other TouringMachines present in the TouringWorld. This model is then used to predict the future actions of the other agents, allowing the anticipation of future states. In the CogAff schema, this modelling layer would probably be incorporated into the deliberative layer.

The TouringMachine architecture was fully implemented and tested [Ferguson, 1992, Chapter 8]. The TouringMachines operated successfully in their TouringWorld environment. There are a number of reasons for their success. Similarly to MINDER1, the concurrently active layers allow the agents to be responsive to environmental change whilst deliberating. A second reason for their success is their modelling layer. There are two facets of the TouringWorld that make the modelling layer successful. The first is that because everything is designed around the driving domain, there are only a fixed number of possible things that an agent can do (e.g. turn left, speed up, stop). This makes predicting and modelling the behaviour of other agents a lot easier, and hence more successful than in a less constrained environment (e.g. the nursemaid world). The second aspect of the TouringWorld that makes modelling successful is that all of the TouringMachines are different instances of the same agent. This allows the agent's modelling layer to use its own decision making processes to build accurate predictions of other agent's future actions. The only thing that individuates the inner processing of different agents is their different long-term goals (destinations in the driving domain). This difference adds uncertainty to the modelling process, although this uncertainty is not as great as it would be if heterogeneous agents were present in the agent's world.

There would be a number of problems if we attempted to directly apply the TouringMachine architecture to create computer game agents. Most of these would stem from the change in operating environment, as the underlying architecture is suitable. A typical computer game has a lot fewer restrictions on possible actions and motivations than the driving domain. Once the restrictions of the driving domain are lifted, the modelling layer would probably be less capable of producing an accurate model from which to predict the players behaviour. Opponent modelling has been previously attempted in the computer game *Quake2* [id Software, 1997] within Laird's work on the Soar Quakebot [Laird, 2000]. From the evidence presented, this application of opponent anticipation was successful in the domain of a deathmatch. There are a great number of similarities between these two domains in which anticipation has been successful. Both domains almost exclusively involve agent positioning and movement, with *Quake2* adding only the extra complexity of weapon selection. This shows that despite fears about complexity, anticipation and opponent modelling can be applied to computer games, as long as the domain remains relatively simple. If the domain was made to be a great deal more complex, then it is possible that the technique would be less effective.

Another problem that would arise if the TouringWorld research was applied directly to a games world is that the variety of possible behaviours required from the agent would be greatly increased. All of the problems in the TouringWorld are of the "get from A to B" variety. These are still present in computer games, but there are also a number of other problems that need to be dealt with by the agent (e.g. managing resources, solving puzzles, hiding from aggressors etc.). Dealing with these extra problems would possibly require alterations to the TouringMachine architecture (e.g. an expanded deliberative layer).

7.2.3 The Guardian agent

The Guardian project [Hayes-Roth, 1990] is another project that has designed an architecture for agents in a real-time, dynamic environment. The architecture (see Figure 7.2) is based on the three tower model[2] [Nilsson, 1998, Chapter 25]. The three towers are situated within a communications interface used to pass messages between the towers. Direct links are present between the perception and action subsystems, facilitating event driven reactions (reflexes). The cognitive subsystem performs any deliberation required by the agent. The work on the Guardian agent goes beyond just presenting an architecture for a real-time agent. It also describes a set of requirements that are desirable for an agent to act successfully in a complex environment. These are:

- **Communications**: The components of an agent's architecture must have the appropriate communications between them.
- **Asynchrony**: The agent must function asynchronously with respect to the environment.
- **Selectivity**: The agent must determine whether and how to perceive, reason about, and act upon various environmental events.
- **Recency**: The utility of reasoning and sense data degrades with time.

[2] Although Figure 7.2 does not appear to have three clearly defined towers, the one-way cycle of information from sensors to perceptual processing to actuators is clearly designed in this mold.

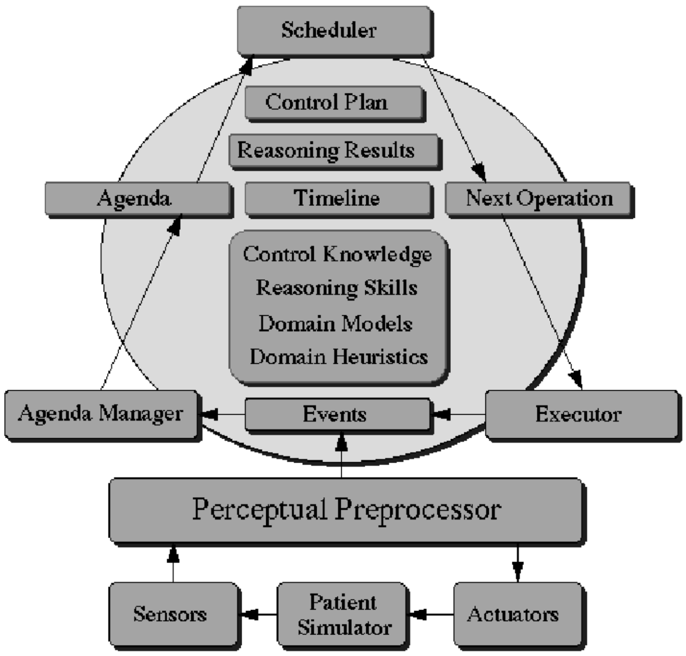

Figure 7.2: The Guardian Agent Architecture [Hayes-Roth, 1990].

- **Coherence**: Where preferable, the agent should produce a globally coordinated (i.e. goal-oriented) series of events.
- **Flexibility**: The agent must be able to react to important unexpected environmental changes.
- **Responsiveness**: A more urgent situation should produce a quicker response than a less urgent one.
- **Timeliness**: The agent must meet various hard and soft real-time constraints on the utility of its behaviour.
- **Robustness**: An agent must adapt to resource-stressing situations by gracefully degrading the utility of its behaviour.
- **Scalability**: The satisfaction of previous criteria should be unaffected by problem sizes.
- **Development**: An agent must exploit new knowledge to improve the utility of its behaviour.

The Guardian agent has been built to satisfy these criteria. To enable it to do this, the architecture introduces three novel concepts; the cognitive satisficing cycle, dynamic control plans, and perceptual filtering.

The cognitive satisficing cycle produces the behaviour in the cognitive tower. The cycle has three elements that operate in a closed loop; the agenda manager, the scheduler and the executor. The agenda manager identifies every action that is possible in the current situation. This list is passed to the scheduler which selects the best action from it. The selected action is then passed to the executor which fills in situation specific details (e.g. variable values) and then executes it. The work identifies the agenda manager as a source of real-time response problems. This is because compiling an exhaustive list of possible actions is an extensive and time consuming exercise. To this end, the processing of the agenda manager is designed to satisfice (i.e. only partially satisfy), rather than completely fulfil its operating criteria. To do this, the agenda manager identifies possible actions until some cycle parameters are met. The cycle parameters heuristically define a stopping point for the search of the agenda manager. They may be based on internal processing or external event criteria. This alteration to its operation allows the agent to better fulfil the flexibility, responsiveness and timeliness criteria (i.e. the real-time performance criteria). The idea of using cycle parameters to define stopping conditions for processing is related to the idea of using an anytime algorithm for the actual processing. If the stopping conditions are relatively course-grained with respect to the occurrence of solutions then an anytime algorithm may not be necessary. On the other hand, if the stopping conditions could possibly occur at any point in the process, then an anytime approach to processing may be necessary.

The Guardian agent uses dynamic control plans to reason about action selection, and guide action selection within a global framework. A control plan is a list of control decisions that influence the Guardian agent's scheduler. Control decisions represent guidelines for what action to take when, policies regarding resource limits and, most importantly, preferences toward reasoning methods that meet current time constraints. A hierarchy of control plans can exist. This allows long-term control to be represented by one plan, with other smaller sub-plans guiding decision making for specific short term events. For example, an agent may have a (control) sub-plan to change its behaviour whilst driving or playing a sport. Control plans are generated by the agent in the same manner that actions are selected. This allows the agent to alter its control plan every loop of its cognitive cycle, but only if the criteria of its current control plan allow it. Although the methods by which control plans are generated are not given in detail, the concept they represent is an interesting one. They are a conceptually different method than the standard mix of reactive actions and deliberatively generated plans we have seen previously. They represent more of a frame-like method of behaviour generation [Minsky, 1997]. Frames are used to store knowledge about situations. When an agent is in a particular situation it retrieves the relevant frame to inform it how to behave, what to expect from the situation and any other facts that may be important. Dynamic control plans are similar to frames in that they guide the whole of the agent's behaviour and that they can refer to specific situations (although general plans are also possible). Dynamic control plans help the agent satisfy the selectivity, coherence, responsiveness, and timeliness criteria set out by the author. They help to satisfy the responsiveness and timeliness criteria by allowing different reasoning methods to be selected. Selection is based on the different time requirements of the methods, which are compared to the time restrictions imposed by the environment.

To ensure that only relevant sense details are passed onto the cognitive and action systems, the perceptual system filters the incoming data. The filter criteria can be adjusted by either the perceptual system itself or the cognitive system. The perceptual system adjusts the filtering based on the rate of flow or change of the data (quantitative properties). The cognitive system adjusts the filtering based on the agent's current reasoning tasks (qualitative properties). This filtering frees the agent from dealing with the full complexity of the world, therefore the amount of processing necessary in all subsequent manipulations of the sense data is reduced (cf. the ideas of dynamic world modelling proposed in [Wood, 1993] and reviewed in Section 3.2.4). This helps the agent satisfy specifically the criterion of selectivity, but also aids the satisfaction of the other time-related criteria (due to the reduction in processing). Sense filtering combined with the agent having limited capacity internal communication buffers poses a serious problem to the Guardian architecture. Information has to potentially pass through a large number of filters before it can affect the processing of the agent. This leads to the possibility that, if the filtering standards are set too high, important facts may get overlooked. This may occur if the environment was in a state of great activity, or if the agent had purposely decided to focus on a particular reasoning task at the expense of others (in this case then this effect may be an accepted risk). This point is recognised by the author: "Allowing the possibility of occasional, more or less consequential error is a necessary concession toward [effective management of complexity]" [Hayes-Roth, 1990]. This may be true in some domains (possibly computer games), but in the chosen implementation domain for the Guardian agent (an intensive care monitoring system) occasional errors may contribute very unfavourably to the agent's utility (not to mention the health of the patient being monitored). Despite this flaw, it seems essential that agents in complex domains are selective about what sense data they use, or else they risk being overwhelmed by amount of information available to them.

7.2.4 The Cypress agent

The Cypress agent architecture [Wilkins et al., 1995] is an architecture based on the existing technologies of SIPE-2 (a planner) and PRS-CL (a successor of PRS [Georgeff and Lansky, 1987], which was reviewed in Section 3.2.2). Its basic operation is much the same as the systems reviewed previously. It features an executor (the reactive component) that continually monitors and reacts to the environment, and a planner that generates new plans for novel situations. These are implemented using PRS-CL and SIPE-2 respectively. The planner is invoked by the executor if it does not have a suitable (reactive) plan to achieve the current objectives. Planning is only done to a set level of abstraction (the *interface* level[3]), then the execution details are filled in by the reactive system.

A part of the Cypress architecture that needs to be examined is how the executor and planner are integrated. Because both are built from existing non-related systems, they are based on different representations. To overcome this problem a mediator language, ACT, is used. Using ACT it is possible to represent knowledge used both by the reactive and deliberative system. The use of the ACT formalism represents a saving in development time at the expense of the speed of execution. A language to translate between two disparate systems allows the developers to quickly integrate the systems. When the agent is actually running, the extra translation work needed to get information to both systems will make the agent's processing slow in comparison to an agent with a similar internal structure that used a common representation language. Also, it is debatable whether both components should be using the

[3][Wilkins et al., 1995] introduced the term "interface level", but in this thesis it was first discussed in Section 3.2.4.

same knowledge representation (see Section 3.2.4).

The SIPE-2 planner is the main feature that distinguishes the Cypress architecture from the other architectures reviewed previously. This planner provides powerful deliberation abilities (e.g. parallel actions, replanning, large detailed plans) that previous architectures have lacked. The only problem is that such a system may take too long to find a suitable plan in a dynamic environment (an environment it wasn't designed for). This will not be too threatening to the agent as it can still react whilst planning is proceeding. However, it may reduce the utility of the plan that is produced. This is summed up by a quote from [Howe et al., 1990]; "the time required to decide what to do must be less than the time taken by the environment to affect changes that render the intended action inappropriate or inexecutable".

7.2.5 An architecture from the Oz project

Research reported in [Blythe and Reilly, 1993] demonstrates another attempt to integrate reactive and deliberative processes in an architecture for an intelligent agent. The reactive element of the architecture uses the HAP system, a descendant of Firby's RAP reactive planner [Firby, 1987]. The planner used is an adapted instance of the classical planner Prodigy. Not a lot of architectural detail is given, but the work does present an interesting approach to the important issue of real-time planner response. A timely response is guaranteed by the planner because it has been altered to be an anytime algorithm. This has been done for the same reason that was identified earlier in this thesis; the agent will have a variable and unpredictable amount of time available to it to deliberate, and it cannot afford to deliberate extensively when a response is required. The approach taken to anytime planning in this instance was reviewed in Section 4.4.2.

The literature only discusses an unfinished version of the agent. One notable issue that is yet to be resolved in the agent is the ability to do anything else whilst planning. Because of the lack of concurrency, the agent cannot perform any other action while this planning is going on, and hence freezes. Rather than being a weakness of this research (as it just requires further implementation), it highlights the need for intra-agent parallel processes in a dynamic environment (echoing the asynchronous activity criteria from Section 7.2.3). Components processing in parallel feature in most designs based on the CogAff architecture schema.

To deal with the agent's dynamic environment (something which the original version of the agent's planner was not explicitly built for), the previously mentioned technique of goal-subgoal instability is used. The planner plans with an abstraction of the current world, this plan is then executed by the reactive system. The reactive layer executes the abstract plan using HAP reactive plans. The abstraction used by the planner eliminates all the dynamic detail of the world, allowing plans to still be built under the static world assumption. This is taking advantage of Wood's concept of goal-subgoal stability by only planning to the appropriate interface level (see Section 3.2.4).

7.2.6 The SOMASS system

The SOMASS system was designed to run an assembly robot [Malcolm, 1997]. As such, it is placed in a domain dissimilar to many of the agents reviewed above (although parallels could be possibly drawn with the domain of the Guardian agent). Uncertainty is still present in this domain as the shapes and positions of the pieces to be assembled are unknown. The real-time behaviour required from the agent involves manipulating these pieces into their correct positions, but there is no time limit on this. The SOMASS system has two (physically) distinct parts, a deliberative assembly planner and a behaviour-based (i.e. reactive) assembly agent. The assembly planner determines how the given parts should be put together to create the end product and the assembly agent acts out this plan to construct the end product. Taken together (as their overall success is utterly dependent on the performance of both components) these components can be viewed as a hybrid agent architecture. As an architecture its composition is similar to the architectures that we have seen previously (a deliberative planner generating behaviour that is executed reactively), but with this the author is more explicit on the advantages of separating processing into reactive and deliberative layers[4].

In [Malcolm, 1997], the author states that by combining behaviour-based and deliberative processing methods in a hybrid agent, the problem domain can be deconstructed into two distinct areas; a problem area that can be tackled from the bottom up, and a problem area that can be tackled up from the top down. The behaviour-based control offered by reactive execution systems is intended to solve problems from the bottom up as it is expressly data driven. Problem solving through planning is useful for top-down problem solving as it can manipulate more abstract information. The developers of SOMASS state that having two parts to the overall problem solving process allows both the developer and agent to avoid having to tackle particular problems with unsuitable methods. More specifically it means that the behaviour-based techniques can deal with the problems featuring unpredictability and the planner can deal with problems involving long term goals. This approach has been taken by other authors (e.g. [Aylett et al., 1995]), and is further evidence of the advantages of creating a divide between knowledge characterised by goal-subgoal instability and that characterised by goal-subgoal stability.

The developers of the SOMASS system also make explicit another reason for decomposing the problem in this manner. By forcing the planner to be ignorant of execution issues (i.e. by only planning down to a suitable interface level) the complexity of the planning problem is greatly reduced. The necessary increase of complexity in the execution system needed to deal with the (execution) issues ignored by the planner is a lot smaller than the reduction in complexity from the planner. This shows that by having the appropriate components in an architecture, an agent developer can make solving certain problems a great deal easier. This information from the SOMASS system also gives an insight into why the other architectures reviewed have approached agent development in a similar way.

[4]The SOMASS system does not use this terminology, but the processing approach they discuss maps transparently onto the CogAff architecture schema.

7.3 Past Computer Game Agents

To add to the information gained by reviewing existing hybrid agents, we will now examine two previous agents that have been developed by the academic community specifically for modern computer games.

7.3.1 The Soarbot project

The Soarbot project is a project aimed at creating an agent for computer game worlds. The specific goal of the Soarbot project (from the University of Michigan Artificial Intelligence Laboratory) is to develop agents (named Quakebots) to participate in the game *Quake2* [id Software, 1997]. The agents are developed using the Soar architecture [Laird et al., 1987]. *Quake2* is from the "first person shooter" genre of games. The Quakebots are designed for deathmatch play. Deathmatches are duels between two or more individuals or teams. The aim of the these games is to reduce your opponent's health until they have none and "die". When players die, they are regenerated at set positions, and the opposition gain an advantage. Deathmatches are typically played over the internet or a local network between groups of players, though sometimes these players are joined by intelligent "bots" (to make up the numbers). It is these bots that the Soarbot project aims to develop.

Because the Quakebot is based on the Soar architecture, it inherits the strengths and weaknesses of this architecture. Soar is a system for managing large numbers of reactive rules in a structured manner. Being rule-based provides the speed needed for the bot to operate in a computer game. The downside of this is that the bot's behaviour is determined by seven-hundred and fifteen rules, an amount that will render debugging behaviour a tricky proposition. The matching problem presented by this volume of behavioural rules (a problem that may hinder other reactive systems) is controlled by using preferences that direct the matching in appropriate directions (rather than matching naïvely). This prevents a possible combinatorial explosion due to problem complexity. Soar rules cause the proposal, selection and application of operators to the current world state. Operators represent actions that can be performed within the game world and can be either primitive (actions can be performed immediately) or abstract (containing more operators). The Soarbot is a purely reactive agent and as such can be considered to be similar to (although much more advanced than) the Pengi agent [Agre and Chapman, 1987] and many other behaviour-based, reactive approaches. As we have seen previously (see Section 7.1) this class of agent succeeds at responding to environmental events in a timely manner, but has more difficulty achieving complex goals with less obvious preconditions.

The Soarbot group have performed a pilot study to investigate how well their Quakebot performs against a group of human players with varying skill levels [Laird and Duchi, 2001]. The aim of the study was to see how the various parameters that can be set in the bot affect its "humanness" and perceived skill level. The parameters that can be varied in the Quakebot are decision time, aggressiveness, complexity of tactics, and critical skill level. The decision time determines how many decision cycles Soar runs a second. Aggressiveness determines the likelihood of attacking options being selected. Complexity of tactics places the bot in one of three skill bands (novice, medium or high) that reflect the range of tactics the agent has access to. The critical skill measured by the study was the bot's aiming ability.

The pilot study produced the following results. As is predictable, the perceived skill level increased as all the Quakebot's parameters were taken to their highest settings. Most significant in this was the bot's aiming skill. Least significant appeared to be the aggressiveness parameter which was cited as having little or no effect on any of the tests performed as part of the study. Interesting results were produced when evaluating the humanness factor. Human decision making time has been approximated at one tenth of a second, and the Quakebot which took decisions at this rate was judged to be the most human-like. The Quakebot with the highest tactical ability was pronounced most human. The hypothesised reason for this is that a more complete set of behaviours lead to more human-like behaviour. The Quakebot with the highest level of aiming skill was recognised as the most un-human. This was because the skill demonstrated was super-human, and hence easily recognisable as artificial.

7.3.2 The Excalibur project

The Excalibur project is a research effort with the aim to produce an agent that can act intelligently in a real-time, dynamic, complex game-like worlds [Nareyek, 2002]. In short, the aims of the Excalibur project are the same as the aims of this thesis. These similar aims have led the Excalibur project to approach the problem in what appears to be a similar way when viewed at a high level. The project has developed an anytime planner to drive its agent (for similar reasons as were outlined in Section 4.2). This planner, and the technology behind it, has already been reviewed in Section 4.4.2. This review will just cover the way the Excalibur project is attempting to build a complete agent using the anytime planner.

Where the Excalibur agent differs from the agent approaches reviewed previously is in its lack of an explicit agent architecture and the absence of hybrid problem solving methods. The anytime agent that the project is developing is based purely on the anytime planner, without any additional processing methods. This aims to take advantage of the "continuous transition from reaction to planning" [Nareyek, 2002] that is potentially provided by anytime planners (i.e. an appropriate reaction for an appropriate amount of processing time). Although this approach is potentially effective, it would not appear to be a robust enough approach to apply to a highly dynamic and unpredictable world. To deal with *any* new situation requiring attention (e.g. a sudden and unexpected threat) the Excalibur agent must create a new planning goal and process, then plan for as long as possible before executing the subsequent plan. For a lot of situations this process will be wholly inefficient, especially when a set of reactive behaviours would do the job faster and possibly more effectively. To deal with events beyond the agent's control, the Excalibur project have developed an extension to the anytime planning algorithm their agent is based upon [Nareyek and Sandholm, 2003]. The extension aims to model the effects external events have on the agent's plans during the planning process. This allows the agent to avoid having its plans invalidated by these potential actions. In some cases this approach even allows the agent to take advantage of the actions of other agents in its world. In terms of dealing with a dynamic and possibly hostile world this approach falls short of what is necessary. This approach only deals with events external to the agent that can be accurately predicted during the design process (this may be acceptable depending on the game world), but gives no consideration to unpredicted events that may require a specific response.

The combination of an anytime planner and a set of reactions may seem inefficient, particularly because anytime planners are used specifically to ensure that a reaction to the environment can be made when it is required. Although this may be the case in some domains, the work on the SOMASS system (reviewed previously in Section 7.2.6) suggests another reason why reactions should be considered in addition to an anytime planner. The reason is that reactive or behaviour-based actions are better than other approaches (e.g. planning) at solving particular problems. If a fast, simple reaction is required then a reactive behaviour will more than suffice. If an extended, goal-directed response is required to an environmental event, then higher-level approaches (e.g. planning) are more appropriate.

7.4 Summary

In this chapter we have seen a number of previous approaches to designing the kind of agent we are aiming to produce in this thesis. We first looked at purely reactive approaches, and although they offered a number of advantages related to processing speed (especially when reacting to environmental events) their lack of guaranteed goal-directed behaviour made them impractical for generating such behaviour in a game-like world. After this the CogAff architecture schema was introduced as a general framework which can be used to guide the development, or comparison of agent architectures of varying complexity. This was followed by a look at a number of hybrid agent architectures and specifically which of their features affected their performance in real-time, dynamic domains. A critical factor in a number of the architectures was having a planner to produce high-level goal-directed behaviour plans, combined with a reactive system to execute them (and fill in missing details). A behaviour-based reaction system to handle sudden, unpredicted events also gives an agent an advantage over agents that can only act on the results of deliberation. After this look at what make agents successful in game-like domains, two of the most prominent academically produced game agents were examined. Although both had features that provide some advantages in game-like worlds, neither had all of the necessary features to make them applicable in a wide variety of game-like domains.

Chapter 8

Development of a Computer Game Agent

This chapter presents the development of an agent designed to behave intelligently in a computer game world. The sections in the chapter represent iterations of the design-based methodology presented in Section 2.1. The iterations represent attempts to refine the description of the problem we are trying to solve (this maps out the agent's *niche*). The revisions of the problem definition are followed by attempts to produce agent designs (architectures in this case, as discussed in Section 2.5) that satisfy the new problem definitions (thus exploring the *design space* of possible agents).

Although we have already specified certain components of the final architecture (most notably the anytime planner which was the subject of a number of the previous chapters), we will start our exploration of design space from a point of purposefully limited knowledge about the problem. This will allow us to witness more convincing and demonstrative arguments for our resulting design decisions.

The four iterations that will be passed through in the design process are as follows;

- Iteration One, Section 8.1. A basic, naïve design is produced as a starting point for the design process.
- Iteration Two, Section 8.2. The basic agent is given mechanisms to monitor its environment for important events and interrupt plan execution processes if events make them unnecessary.
- Iteration Three, Section 8.3. The agent is then given similar abilities but with the focus on planning rather than plan execution.
- Iteration Four, Section 8.4. The agent is finally given full anytime planning abilities and the mechanisms to support this.

8.1 Iteration One: The Basic Agent

In this first iteration we want to design an agent that will overcome the problems that have been discussed throughout this thesis (and first presented in detail in Sections 2.3 and 2.4). These problems represent the *niche* into which the design should fit. The design process will be driven in part by the knowledge gained from the review of previous attempts to design such agents. It will also be driven by the desire to explore the possibilities offered by new designs.

As presented previously, the design of the agent architecture will represent a subset of the CogAff architecture schema (see Section 7.2.1). By taking this approach we are immediately limiting the possible designs we can produce, but due to the very general nature of the CogAff architecture schema, we are only limiting ourselves to a sensible and previously justified subset of all possible designs.

8.1.1 Design

The basic construction of the agent can be seen in Figure 8.1. The thick lines outlining the architecture represent segments of the CogAff schema that are being utilised. The dashed lines represent segments that are currently unoccupied. Within the boundaries of the architecture, the curved boxes represent the components that make up the architecture, and the lines between them represent the (directed) flow of information. The architecture has a number of the features that have been discussed previously in this thesis. These will be discussed in the followed sections. Each component in the architecture will have a section dedicated to it for each design and implementation iteration where necessary. In addition to this, sections will also be dedicated to important design issues not depicted on the architecture diagram (e.g. the goals and knowledge representation used by the agent).

8.1.1.1 Communication and teamwork

Although many computer games (including the Capture The Flag scenario) are team games, the design for the agent will generally avoid issues involved with communication and coordination between multiple agents. These issues are avoided because they (particularly coordination) are hard tasks that generally require fairly focused development efforts. As the main focus of this thesis is on other issues, communication and coordination will be overlooked except where totally necessary.

8.1.1.2 Goals

At any point in time the agent will be pursuing a number of goals. One of these will be an explicit goal whilst the rest will be implicit. Explicit goals will have a representation within the agent and will be used to guide the agent's planner which in turn will produce a plan for the agent to follow to achieve the explicit goal. For example, in the Capture The Flag (CTF) scenario explicit goals include scoring a point (capturing the flag) and intercepting an opponent who has stolen your flag.

Implicit goals do not have a representation within the agent, but instead are encoded into the behaviour of its components. For example, in the CTF scenario implicit goals include winning the game and staying alive. These goals do not have explicit representations, but rather they cause other behaviours to occur (e.g. reactions to keep the agent out of danger) and other goals to be generated (e.g. the implicit goal to win the game causes the explicit goal to score a point to be generated).

Each of the agent's explicit goals represents a goal that can be planned for by the agent's planner. If the example agent had explicit goals that couldn't be planned for, then they could never be achieved

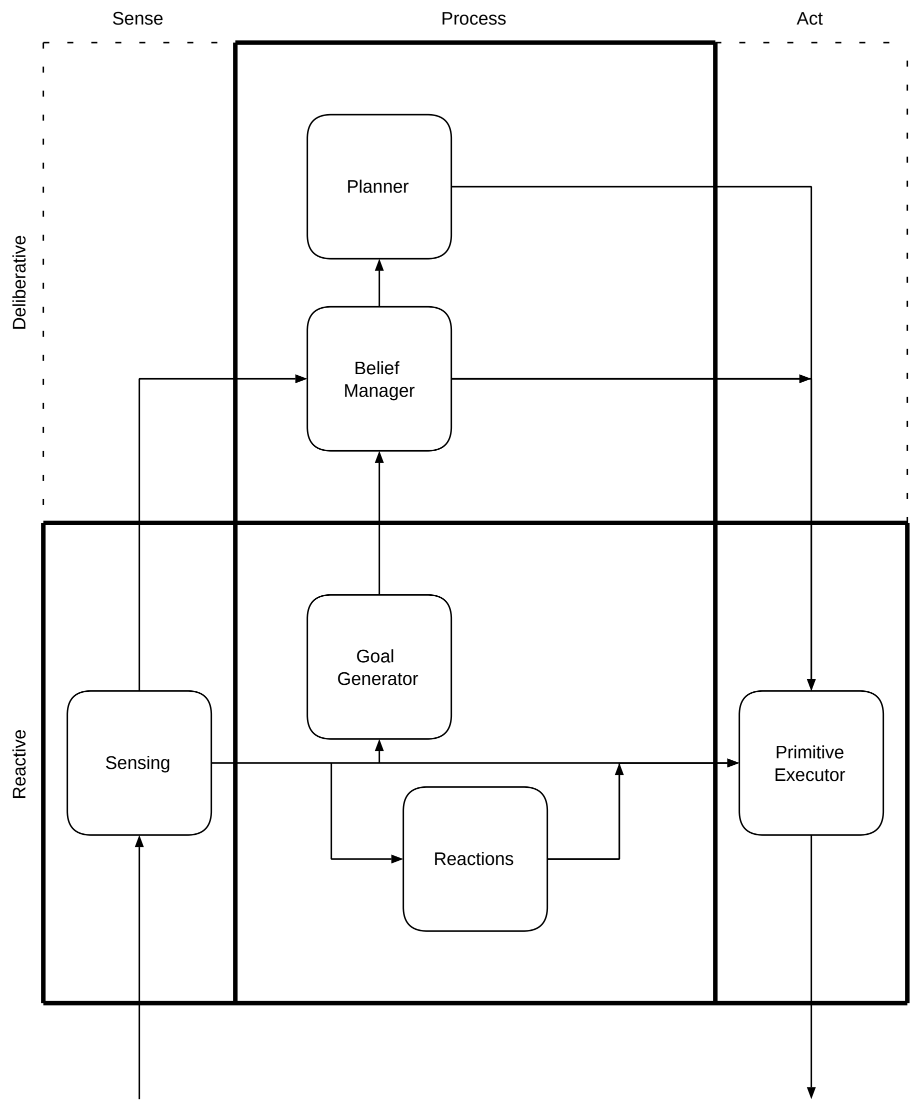

Figure 8.1: Agent Architecture Design For Iteration 1. This architecture represents the most basic components necessary for the agent to fulfil the requirements of its niche.

through "conscious" action by the agent. This also means that the agent's execution component must allow the agent to execute at least the actions required by the planning primitives. If it doesn't, then there is no way that the agent can execute its plans.

To ensure that it is always behaving in an appropriate way for the current state of the environment, the agent (or more specifically, the belief manager) should associate each explicit goal with an importance judgement. In design terms, it is not important how this judgement is determined by the agent, but just that the agent can determine whether one explicit goal is more important than another explicit goal. The importance judgement is intended to reflect how important it is for the agent to attempt to achieve a given goal relative to other goals that may have been proposed. Another way to consider importance is that achieving a more important goal offers a greater utility than achieving a less important one.

8.1.1.3 Sensing

The sensing module for the agent translates external events into a representation that can be used by the agent. This representation is then accessed by almost all of the components in the architecture. Because the resulting data is used so widely, it is necessary to construct it in a form that can be easily manipulated and updated. This allows the agent to carry out any internal processing without having to do much additional translation. The choice of what actual representation to use should be determined by the exact processing that must be done on the data, and the development environment in which the agent is implemented.

8.1.1.4 Reactions

The first of the two critical parts of the reactive processing layer is the module controlling the agent's reactions. The purpose of this module is to continually monitor the incoming sense data and trigger appropriate actions if the agent is placed in a situation where an immediate response is required. The reaction module does not control the actions itself, rather it causes the agent's execution component to execute a particular action. Which situations cause which reactions are usually determined by the implicit goals of the agent.

The reaction module is required for a number of reasons. First, because most of the agent's behaviour will be generated by a deliberative planner, it must have a way to respond quickly to environmental change either when no plan is available, or when a plan is being executed and the current situation requires a response that is not present in the plan. Second, even in situations to which the agent's behaviour is perfectly suited, the environment can change so quickly and unpredictably that the change cannot be anticipated by the agent. In such situations, reactions are invaluable.

8.1.1.5 Goal generator

The *goal generator* is the architectural component that provides the primary source of motivation for the agent. Its role is to continually monitor the state of the world (via sense data) and propose the goal that the agent should be trying to achieve at the current point in time. The type of goals being proposed by the goal generator are the explicit goals that the agent can pursue (such as scoring a point, or going to find some health in the Capture The Flag scenario).

The goal generator should run continually and suggest goals without any knowledge of the agent's current or previous behaviour. As such, the agent architecture needs to provide another component to filter this stream of goals based on its current behaviour. This design is based on the behaviour of the variable filter mechanism proposed in [Wright, 1997] and reviewed previously in Section 7.2.1.

The reactivity of the goal generator is critical to the real-time performance of the agent. If the agent was continually performing complex reasoning (e.g. some kind of inference) to determine what it should do next, it would risk falling behind its environment if the environment changed faster than it could reason. By generating goals reactively, the chances of this happening are greatly reduced. The agent can never truly eliminate the processing needed to determine its next course of action, but it is better (faster) to perform this processing in two discrete steps (goal generation and goal reasoning) than in one step (reasoning about which goals are possible then which of the possible goals should be chosen).

Reactive goal generation is possible (particularly in game worlds) because there is usually a limited set of possible explicit goals that the agent can achieve, and each goal is usually only applicable in a small number of situations. So, rather than implement the logic required to deduce a limited set of goals from a limited set of circumstances using a (slow) deliberative system, they can be indexed in a (fast) reactive system. If the situations were not limited, then this may not be either possible, or as effective.

8.1.1.6 Primitive executor

The *primitive executor* is the component that the agent uses to affect the world. It takes input plans produced by the planner, combines them with beliefs about the world from the belief manager, then acts out the plans in the agent's world. The primitive executor also takes sense data as input to allow it to coordinate actions with the current state of the world. The concept of a primitive executor has been seen in many of the previously reviewed architectures, e.g. the Cypress agent reviewed in Section 7.2.4.

The primitive executor is important in the architecture because it allows a division to be made between reasoning knowledge and execution knowledge (cf. the SOMASS system reviewed in Section 7.2.6). Execution knowledge involves the current state of the world, including the positions and speeds of objects (in physically or virtually embodied agents). This kind of knowledge changes frequently and it is critical that the agent has access to its most recent values if it is to execute actions correctly. Reasoning knowledge is typically more abstract than execution knowledge and is used for high level decision making within the agent. Reasoning knowledge typically includes object types, relationships between less abstract knowledge items (e.g. something is near something else) and the results of previous reasoning processes (e.g. plans or other types of new knowledge). It is the role of the primitive executor to receive reasoning knowledge (e.g. an action from a plan) as input then affect the agent's environment in the way specified by this knowledge whilst taking into account all relevant execution knowledge (the state of the agent's environment).

The need for a distinction between types of knowledge that have different rates of change was first addressed in this thesis when the work on the Autodrive project was reviewed in Section 3.2.4. In these terms, reasoning knowledge is characterised by goal-subgoal stability and execution knowledge is characterised by goal-subgoal instability.

If the architecture didn't have a component to carry out the translation from reasoning knowledge to execution knowledge, one of two possible situations would have to occur. The first possibility is that the actions produced as the result of reasoning processes would have to be directly executed in the world. This would result in a degree of latency between the agent's processing (what it is "thinking") and its actions (what it is doing) because the reasoning knowledge could be out of date. The second possibility is that the agents reasoning would have to encompass execution knowledge. The problems with this are are twofold. First, this would result in an increase in complexity for the reasoning process. Second, as with the previous possibility, by the time the agent attempts to execute its behaviour, the execution knowledge used as the input to the reasoning process will probably be out of date.

The behaviour of the primitive executor for an agent in a real-time, dynamic world is critical. This is because the agent can only affect the world through its actions. Also, the nature of the planner and the plans it produces is important as it affects exactly what the executor must do (cf. [Aylett et al., 1997]). From our previous analysis of the domain, we can identify two requirements for the executor.

The first requirement is that the executor can take planning primitives and accurately translate them into game-world behaviour that fulfils the primitive's purpose. Without enforcing this requirement, the agent would not be able to achieve the goals that its plans had been created for because the plan primitives would not establish the necessary states in the game world. To satisfy this requirement, the behaviour of the primitive executor must be goal-directed.

The second requirement for the behaviour of the primitive executor is that it must be able to perform effectively in a real-time, dynamic environment. The primitive executor might have to deal with tracking fast moving objects, unexpected encounters with other moving objects or navigation around unexpected obstacles.

8.1.1.7 Belief manager

The *belief manager* provides a lot of the control within the agent, stores data for other processes and does the majority of the non-planning reasoning. The belief manager provides control in two ways. First, it acts as the mechanism to filter the goals suggested by the goal generator. To do this it examines how important the suggested goal is. If the suggested goal is more important than the goal currently selected for planning or execution, it replaces the currently selected goal with the newly suggested one. If the suggested goal is less important, then it is ignored. This relies on no goals having an identical quality (although if such a situation was unavoidable, then recency or time already invested could become an important factor). This design for a goal filtering mechanism is quite simplistic, but will suffice for the current computer game context. Extensions to the goal filtering mechanism could see similar or non-conflicting goals combined to trigger one planning process, or goals of different importance added to a queue.

The second way in which the belief manager provides control is by mediating the agent's sense, plan and act cycle. This is done by examining the agent's current goal and the current internal state of the agent. If no plan exists for the current goal, then the belief manager triggers the planner to create one. If a plan exists for the current goal, then this plan is passed to the primitive executor, along with any necessary domain knowledge.

The data stored by the belief manager are primarily facts about the agent's world that have either been sensed or generated by the agent. The sensed data that are stored are restricted to data that are fairly stable within the environment (e.g. what team an agent is on, and the position of the team bases). Generated facts are either facts about the external world that the agent has generated from sense data (e.g. the opponents usually take one minute to score a point once they have the flag) or facts about its own processing (e.g. the planner usually takes thirty seconds to produce a plan to score a point or the current goal is to find health). These stored facts are used within the belief manager to aid reasoning and decision making (e.g. whether a proposed goal should be accepted) and are also sent to other components as input.

One of the most important roles performed by the belief manager is to use its stored data to construct the initial state for the planner. Because the initial state information required by the planner must be formatted in a particular way (the actual way will depend on the planner being used in the implementation of the design), the belief manager must convert its stored data into the correct format before starting the planning process. This effectively creates a "snapshot" of the current world state, which is then used to plan with. We can again compare this to the ideas of Wood (Section 3.2.4). What the belief manager must do is create a special representation of a changeable world (the initial state for the planner) on which complex reasoning can be performed.

8.1.1.8 Planner

The planner's role in the architecture is to construct an executable plan from the input provided by the belief manager. The inputs are the standard planning inputs of an initial state and a goal to achieve from that state. At this stage the design does not specify the nature of the planner, but it is important that the planner is tailored to its role in the architecture and takes advantage of the primitive executor. To do this it is essential that the planner represents its final plans at the level of abstraction of the interface level (see Section 3.2.4). As we discussed in the design of the primitive executor, to plan in any greater detail than this risks an increase in planning complexity and introducing latency between the agent's processing and its environment. Planning in any less detail than the primitive executor can handle also introduces complexity as the executor would then have to do more processing (using execution knowledge) to determine the correct actions to take.

8.1.2 Example implementation

As seen in Section 2.1, the next stage in the design methodology is to produce an example implementation of the design in order to test how well it fits its niche. The implementation of the agent is written

Goal Description	**Symbol**	**Importance**
Return a dropped flag	`returnflag`	6
Intercept an agent that has the team's flag	`interceptflagcarrier`	5
Protect a flag-carrying team-mate	`defendflagcarrier`	4
Find a way to increase health	`heal`	3
Score a point	`scorepoint`	2
Defend the team base	`defendbase`	1

Table 8.1: Goals used by the agent in the Capture The Flag domain.

to play in the Capture The Flag scenario. It is implemented using Pop11 and the SimAgent toolkit [Sloman and Logan, 1999]. The agent plays Capture The Flag in the commercial game *Unreal Tournament* [Epic Mega Games, 1999] via the Gamebots interface [Kaminka et al., 2002].

The agent's implementation is based on a database of facts and a structured set of condition-action rules that are matched against the database and can subsequently alter its contents. To perform more complex reasoning, rules can either call functions that are external to the ruleset or add temporary data to the database and then pass control to another rule.

8.1.2.1 Communication and teamwork

Capture The Flag is a team game. The agent must coordinate with its teammates to prevent duplication of action and to further the efforts of the team to win the game. Coordination is carried out through simple communication. If the agent is in a situation which it believes its team mates should know about, it sends a message to them containing knowledge that they can add to their databases. This knowledge will then alter the way they process information (principally the goal generation process). Examples of this from the example agent implementation occur when an agent captures the flag and when an agent is under attack whilst holding the flag. When an agent gets the flag it broadcasts this fact to all of its team mates. This is intended to stop them attempting to capture the flag when it is already in the team's possession. When an agent is under attack whilst holding the flag it broadcasts its predicament to its team mates in the hope that they will generate a goal to assist it.

8.1.2.2 Goals

The explicit goals used by the example agent are presented in Table 8.1. Because the agent uses an HTN planner, each goal is associated with an HTN root node from which the planning process reduces primitive plans.

To simplify the implementation, the relative importance of the explicit goals is determined by fixed numerical values. A higher value represents a greater importance. The ordering of the importance values was determined after playing the game and observing how achieving particular goals (and the behaviour required to achieve them) affected the state of the game. This ordering was reviewed and altered during the implementation of the agent[1]. It was determined that preventing the opposition scoring was of utmost importance, so goals that supported this were given priority (returning a flag is the most important because any player can pick up a dropped flag). After this it was determined that helping a teammate that already has the flag is the next most important goal. This is because a lot of effort has already gone into the teammate obtaining the flag, and it is better to support this effort (e.g. attempting to fight off opponents, or being on hand to pick up a dropped flag), than to start from scratch trying to obtain the flag (it is also not possible to obtain the opponent's flag when a team member already has it). The goal to obtain more health is next most important. This is because it is usually ineffective to attempt any of the goals of lesser importance than this with low health. Next comes the goal to a score point, which should be attempted if no other situation requires attention. Finally, the goal to defend the team's base is the least important. This is for two reasons. First, although preventing the opposition obtaining the flag is important, it is often achieved more effectively by attempting other goals that threaten the opposition into more defensive actions. Second, the goal to defend the base is generated as the default goal if no other goal is applicable. If it had a higher importance than other goals, the agent would never leave the base, even if critical events occurred in the game.

8.1.2.3 The interface level

In the design of the agent (and in the previously reviewed literature), the importance of knowledge representation was an issue that was encountered repeatedly. The agent must perform planning and reasoning with information that is stable and reliable, whilst it must execute behaviours using unstable knowledge that is updated as regularly as possible. In the example implementation of the agent design this distinction is made by separating continuous-valued data from discrete-valued data in the agent's representations. Continuous values such as game-world positions and rotations are used by the agent's reactive components (principally the primitive executor and agent's reactions). In the deliberative layer, components deal with much less detailed knowledge. In terms of knowledge representation, the belief manager will typically deal with the names of agents and teams, and the different goals an agent can adopt. The planner also uses this type of knowledge. When a plan or action is passed to the primitive executor, it must first translate the symbols it is given into something concrete in the game world.

A good example of this process in action is how positional data is dealt with by two architecture components with different knowledge requirements; the planner and the primitive executor. The planner must have some knowledge of the positions of certain objects in the agent's environment if it is to construct sensible plans (e.g. there is no point an agent planning to capture the other team's flag when another team member is already holding it), but it is impractical for the planner to use continuous positional data. The impracticality comes from at least two sources. First, there is the additional complexity that comes from reasoning about how to alter and compare real-valued positional data (e.g. how much does a movement

[1] An interesting task may be to determine how these values could be learnt by a future iteration of the agent. One possible way would be to randomly choose an ordering, then evaluate the success of the ordering based on the agent's success in the game.

action alter the agent's position by, and in what direction)[2]. Second, there is every chance that data represented at a coordinate level will be invalid by the time the plan is executed, making its use irrelevant to the final plan anyway. Instead of continuous positional data, the planner uses an `at` predicate, that takes two arguments; the object's name, and the waypoint nearest to the object. Waypoints are invisible points in the game world that the agents use for navigation. An example of the use of such a predicate is `at(Yigal,RedBase)`. When a plan containing such a predicate is passed to the primitive executor, the simplest option would be to substitute the game-world position of the waypoint for the location of the object. This would cause problems for two reasons. First, the object originally only had to be near the given waypoint and this doesn't place the object very accurately. Second, there is no guarantee that the object is still near the given waypoint. Instead of taking this approach, the primitive executor checks the incoming sense data for information about the object. If the object is visible, then it uses its new position data. If the object is not visible then the agent heads for the waypoint. If it subsequently senses the object on the way to the waypoint, it then reverts to using sense data for more accuracy. If the object is not sensed by the time the agent reaches the waypoint, then the agent can explore for the object or fail to execute the plan.[3]

8.1.2.4 Sensing

In the example implementation, the agent's sensors transform incoming sensor strings from the Gamebots interface into data that the agent can manipulate. These data are presented to the agent in the form of facts in its database.

8.1.2.5 Reactions

The agent's reactions are implemented as rules that match against sense data in the database. If a reaction rule is triggered then it adds a fact to the database telling the primitive executor what action to take. For example, if the agent suddenly spots the opposing team's flag carrier, then a reaction rule adds the fact to attack the opposition agent into the agent's database.

8.1.2.6 Goal generator

The goal generator functions much like the agent's reaction component. Simple rules are compared against the current state, and if the conditions of a rule match, then a fact is added to the database suggesting the new goal to pursue. The goal generator can only suggest one goal each time it is run. The goal generator's rules are evaluated in order of the importance of the goals they suggest, so if a goal is suggested by one rule, there is no point in another rule suggesting a goal after this as it would be less important, and therefore unlikely to be adopted by the agent. As the goal generator has no knowledge of

[2] There are planners specifically designed to manipulate real-valued data. For example, SHOP [Nau et al., 1999].

[3] Unless something occurs in the environment to change its behaviour, the agent will explore for a fixed time before giving up.

follow-nodes-to(?position):

$$\begin{array}{rcl} \textit{at(?position)} & \rightarrow & \textit{nil} \\ \textit{know-path-to(?position)} & \rightarrow & \textit{run-to(?next-node)} \\ T & \rightarrow & \textit{plan-path-to(?position)} \end{array}$$

Figure 8.2: An example Teleo-reactive plan for following a list of path nodes.

		Generated By		
Action	**Primitive**	**Plan**	**Reaction**	**Internal**
change weapon	`changeweapon`	Yes	No	Yes
plan path	`planpath`	Yes	No	Yes
follow nodes	`followpathnodes`	Yes	No	Yes
get in range	`getinrange`	Yes	No	Yes
engage	`engage`	Yes	Yes	No
rotate	-	No	Yes	Yes
run to	-	No	Yes	Yes
strafe to	-	No	Yes	No
explore	-	No	No	Yes
follow	-	No	No	Yes

Table 8.2: Actions used by the agent in the Capture The Flag domain.

the agent's current behaviour, it will often suggest a goal that is currently being pursued by the agent. It is up to the belief manager to filter out repetitious goal suggestions.

8.1.2.7 Primitive executor

In the review of reactive agents (Section 7.1) we have already encountered a reactive system that can satisfy the design requirements of the primitive executor. The Teleo-reactive program (TRP) system provides reactive goal-oriented behaviour whilst being responsive to changes in the execution environment [Nilsson, 1994]. The example agent's primitive executor is implemented as a TRP-style system. The input to the executor comes as a list of planning primitives that must be executed in series. Every possible planning primitive has a corresponding Teleo-reactive program, and there are additional TRPs that are used by the reaction component and also internally by the primitive executor. The complete list of actions that the executor can execute can be seen in Table 8.2[4].

Figure 8.2 shows an example Teleo-reactive Program which enables the agent to follow a list of waypoints that form nodes in a path to a particular position. The position variable (*?position*) holds a set of coordinates in the agent's world. The primitive executor substitutes these coordinates for a waypoint or object position as was described in the discussion of the interface level. Paths are stored in the agent as a list of path nodes that must be followed to get to a particular point in the world. In the example TRP,

[4] In the example implementation, path planning is treated as an action because it is performed by an external process (a process within *Unreal Tournament*), and is therefore handled by the agent's executor. If an internal method was used, path planning would almost certainly still be considered an action as it would still have to be performed during the execution of a plan.

if the agent does not have a path to the position *follow-nodes-to* was called with, then it must first plan a path to that position (using *plan-path-to*). If the agent has a list of path nodes (either as a result of calling the path planner, or from a previous action) then it selects the next node to go to and heads towards it (using *run-to*). If the agent finds itself at the desired position (either through its actions or a serendipitous occurrence) then it exits the TRP. Both *plan-path-to* and *run-to* are also TRPs that represent actions that the agent can take.

Two action primitives from the Capture The Flag planning domain (see Appendix A) do not have associated actions in the primitive executor. These are `pickup` and `putdown`. These actions are omitted because they are beyond the agent's control in the *Unreal Tournament* game world. When the agent walks over an object, it automatically picks it up. When the agent gets the opponents flag (the only object the agent would want to put down) back to its base, the flag is dropped automatically.

The actions that are associated with plan primitives all take symbolic parameters (names of agents, objects and waypoints). This supports the division of knowledge between the reactive and deliberative layers in the architecture. The other actions use either symbolic data or continuous data (game-world positions and rotations).

The primitives presented in Table 8.2 present an interesting view of how actions can be represented within a hybrid agent architecture. In the deliberative layer (specifically the agent's planner) when actions such as `planpath` are manipulated they are treated as action primitives. In contrast, the reactive layer (specifically the primitive executor) has a different set of action primitives which it uses to execute the instructions provided by the deliberative layer. This is another effect of separating knowledge along the lines of detail and rate of change. The deliberative layer uses primitives that allow reasoning with stable knowledge (characterised by goal-subgoal stability) and avoid unnecessary increases in complexity. The reactive layer uses primitives that are as close to the observed world as possible. It does this to allow accurate reactions and action executions. It is able to use detailed representations because it does not perform complex reasoning with the information. The reactive layer suffices with pattern matching and some basic comparison operators, whereas the deliberative layer performs planning and reasoning.

8.1.2.8 Belief manager

In the example implementation of the agent, the belief manager controls the sense, plan and act cycle by storing a representation of what the agent is currently doing (planning or acting). This is checked when a decision is required, and then either planning or acting is triggered.

The initial state for the planner is mostly generated by reformatting the agent's sense data, but more work is required in the cases where positional data must be abstracted away. The agent has a list of all of the waypoints in the environment along with their positions[5], and for each fact containing positional data it substitutes the data with the name of nearest waypoint. The planner also makes use of the predicate `near` to determine when one object is near another. As this relationship is not directly observable by the

[5]This could be constructed as the agent explores its environment, but in this instance it has been precompiled for convenience.

agent, the belief manager must calculate which objects in the environment are near which others, then add this information to the initial state[6].

Because the initial state can contain a lot of information surplus to the planning process, the belief manager also prunes this information from the initial state. This pruning is done based on the agent's current goal, and is solely an optimisation to speed up the planning process[7]. The pruning algorithm was developed by looking at all the possible plans that could be developed for a particular goal, then allowing the belief manager to remove facts from the initial state that never feature in the plans for the current goal (either as a positive condition or in a negation). This pruning is done every time an initial state is constructed for the agent's planner, with the information about what to prune encoded in a series of rules in the belief manager.

The belief manager's role as a filter for the goals proposed by the goal generator is simplified by the use of a numerical representation of importance. The proposed goal with the highest importance value is selected as the current goal by the belief manager[8]. In this basic design, the goal being acted upon can only change at two points in the processing cycle of the belief manger. These two points are after a planning process has terminated and after a plan has been executed. Goals that are suggested previous to these points are either ignored (if they fall below the importance threshold represented by the current goal) or retained until they can be acted upon (unless they are superseded themselves).

One of the more interesting jobs done by the belief manager is to mediate between the planner and the primitive executor. Because the planner creates plans using a "snapshot" of the world taken at a previous point, there is a risk that the knowledge it used may no longer be current. To overcome this, the constants in the plans that are produced are treated as variables by the belief manager. Every time a constant is introduced by the planner, it also creates a dependency which links that constant to facts in the initial state. For example, if it introduces a constant representing the red team's flag, it may associate this with the facts that this constant represents an object that is a flag and is at a particular point in the world. Before the plan is passed to the primitive executor, the belief manager creates a new "snapshot" of the world (which contains information that is currently accurate to the best of the belief manager's knowledge) and attempts to unify the dependencies within the plan with this new snapshot. If unification is possible, then the new unified values are substituted into the plan before it is passed to the primitive executor. If the unification is not possible, then the plan is no longer valid and either a new planning process is started or a new goal is adopted.

8.1.2.9 Planner

The planner used in the example implementation of the basic agent is an HTN planner based on the UMCP algorithm [Erol, 1995]. It is effectively the implementation of A-UMCP (the anytime planner

[6]Although generating facts about relationships between observed objects is done by the belief manager in this instance, there is a case for placing this in a separate component for deliberative sensing.

[7]The difference made to planning speed by pruning the initial state of superfluous facts is discussed in Section 6.4.

[8]As was stated in the design for this component in Section 8.1.1, this relies on no two goals having identical importance values.

discussed earlier in this thesis) without its anytime capabilities.

The only addition made to the planning algorithm specifically for its role in the agent architecture was the aforementioned behaviour of recording dependencies for the constants used in the plan. This alteration is fairly trivial as new constants are only introduced during the enforcement of a particular variety of task network constraint. These constraints contain the facts being used to add the variable into the task network, so these facts are simply saved along with a reference to the constant as the dependency for that constant.

At this stage in the design process, the only other relevant information regarding planning relates to its speed. Whereas the agent has been designed to be situated in a complex environment containing a wide variety of information and presenting complex reasoning and planning tasks, the Capture The Flag environment in which the example agent is implemented is very simple. This has made the implementation of the agent easier than it could have been (given a much more complex world), but unfortunately it means that the agent architecture is not being evaluated in the domain for which it was designed. To attempt to remedy this, the planner in all of the example implementations (including this basic implementation) has had its speed artificially reduced to simulate the game world being a great deal more complex than it actually is. This will have the consequence of exaggerating the effect that iterations of the design process will have on reducing the time the agent spends using the planner. Although this is an exaggeration in terms of the Capture The Flag domain, it is indicative of the effects changes in design will have when the agent is situated in a more complex environment. An additional justification for this artificial slowing of the planning algorithm is that it more accurately reflects the processing power that would be available to a game agent in an actual computer game. A planner may be fast when run as a single process on a high powered machine, but if the planner was placed on a much lower powered machine, or if a large number of other processes (including other planning processes) were running at the same time on the same machine, then the speed of the planner would be greatly reduced.

8.1.3 Analysis of example implementation

Once the agent design has been implemented, it must be tested to evaluate how well the design fits into its specified niche. In this case we are interested in the real-time performance of the example agent (as this has been the driving force behind the design process). To examine the performance of the agent, five quantities are measured. These are;

- **Planning time**: The amount of time an agent spends planning during a single game.
- **Wasted planning time**: The amount of time the agent spends planning for goals after the goal is no longer valid.
- **Execution time**: The amount of time an agent spends executing plans during a single game.
- **Wasted execution time**: The amount of time the agent spends executing plans to achieve goals after the goal is no longer valid.

Quantity	Value
Avg. Planning Time	138 secs
Avg. Wasted Planning Time	37 secs
Avg. Execution Time	153 secs
Avg. Wasted Execution Time	95 secs
Avg. Percentage of Goals Selected	24%

Table 8.3: Results from the first iteration of the design process.

- **Percentage of goals selected**: The percentage of proposed goals that are selected and planned for.

These measurements are intended to reflect two qualities of the design. First, the amount of wasted time and the percentage of selected goals reflect how well the agent is keeping up with the goal-affecting changes in its environment. The more time that is wasted, or the smaller the ratio between the goals suggested and selected, the worse the agent is at keeping its behaviour current. Second, these measurements provide an insight into the overall performance of the agent. If the agent is not selecting enough of the proposed goals and is wasting a lot of time, then it is not going to be competitive in the game world (although this is really a side-effect of not keeping up to date with its environment).

It would be desirable to include a measurement that reflects how many goals the agent manages to achieve. Unfortunately, in a dynamic world with other agents attempting to achieve similar and adversarial goals, it is quite hard to determine whether the planned actions of an agent have achieved a goal, or whether it has been achieved through a well timed reaction or through pure serendipity (cf. partial achievement of goals, Section 3.1). It proved easier for the agent to claim causal power over the achievement of some goals rather than others. Notably these were goals that involved finite, less abstract actions (e.g. to get the flag, go here then go here) rather than open-ended, more abstract actions (e.g. to intercept the flag carrier, first find it, then fight with it until you've won). Because the number of attempts to achieve goals that have clear causality did not come close to dominating the overall ratio of goals pursued, it was felt that measuring them would not provide a reliable source of information about the overall performance of the complete agent architecture.

To test the design, two teams of two example agents played ten five-minute games of Capture The Flag in *Unreal Tournament*. This resulted in data from forty separate instances of the example agent. The results produced by the example implementation of the design can be seen in Table 8.3. The table shows that a great deal of time was wasted by the agent both planning for and executing plans for goals that were no longer valid. From examining these results along with the basic agent design, it can be seen that the fact that the agent continues to execute plans once their goals are no longer valid appears to be the biggest source of latency between the agent and its environment. As this is the first iteration of the design process, we have nothing to compare the average percentage of goals selected against. Without a suitable comparison, it is hard to know how this value reflects the performance of the agent. The need for comparison is not as important when analysing the amount of time wasted, as any amount of wasted time reflects a drop in the agent's real-time performance.

8.2 Iteration Two: Execution Interrupts

The previous analysis led to the identification of periods during which the agent is executing a plan when it should really be pursuing a different goal. These periods are times when the agent's behaviour is lagging behind the important events (i.e. those events that cause explicit goals to be generated) that are occurring in its environment. We now must alter the agent design to attempt to overcome this problem. Wasted execution time has been chosen as the problem to tackle because, as identified in the preceding section, it is the principal source of wasted time.

8.2.1 Design

Rather than making wholesale changes to the original design, we must focus on the components in the agent's design that are involved in the problem we want to overcome.

8.2.1.1 Belief manager

It is the belief manager that holds the key to reducing the amount of time wasted during plan execution. In the basic design, when the goal generator proposes a goal that is more important than the one currently being pursued, the belief manager simply stores this goal. Then, when the agent finishes what it is currently doing (either planning or executing) the goal is selected. To reduce the amount of execution time wasted by the agent, we need to redesign the belief manager to immediately halt the execution of a plan if it is discovered that the plan's goal is no longer the agent's current goal (i.e. a more important goal has been proposed). But, rather than alter the behaviour of the belief manager too drastically, we can make a small alteration to it and add a new component to the architecture to handle the rest of the necessary behaviour. Approaching the redesign in this way minimises the risk of damaging the existing behaviour of the previously designed components. The small alteration to be made to the belief manager is to allow it to immediately select a proposed goal to be the current goal if the proposed goal is more important than the current goal. The new component to be added to the agent architecture is the *interrupt manager*.

8.2.1.2 Interrupt manager

The interrupt manager's role within the agent architecture is to monitor the current goal held by the belief manager and ensure that the primitive executor is not executing a plan for any other goal. Its relationship with the rest of the architecture is depicted in Figure 8.3. It takes input from both the belief manager and the primitive executor. From the belief manager the interrupt manager receives input on the agent's current selected goal. From the primitive executor the interrupt manager receives input on the goal associated with the plan currently being executed. If the goal selected by the belief manager is more important than the one supplied by the primitive executor, the interrupt manager generates an interrupt event. This event causes execution to stop and allows the belief manager to select a new goal for planning.

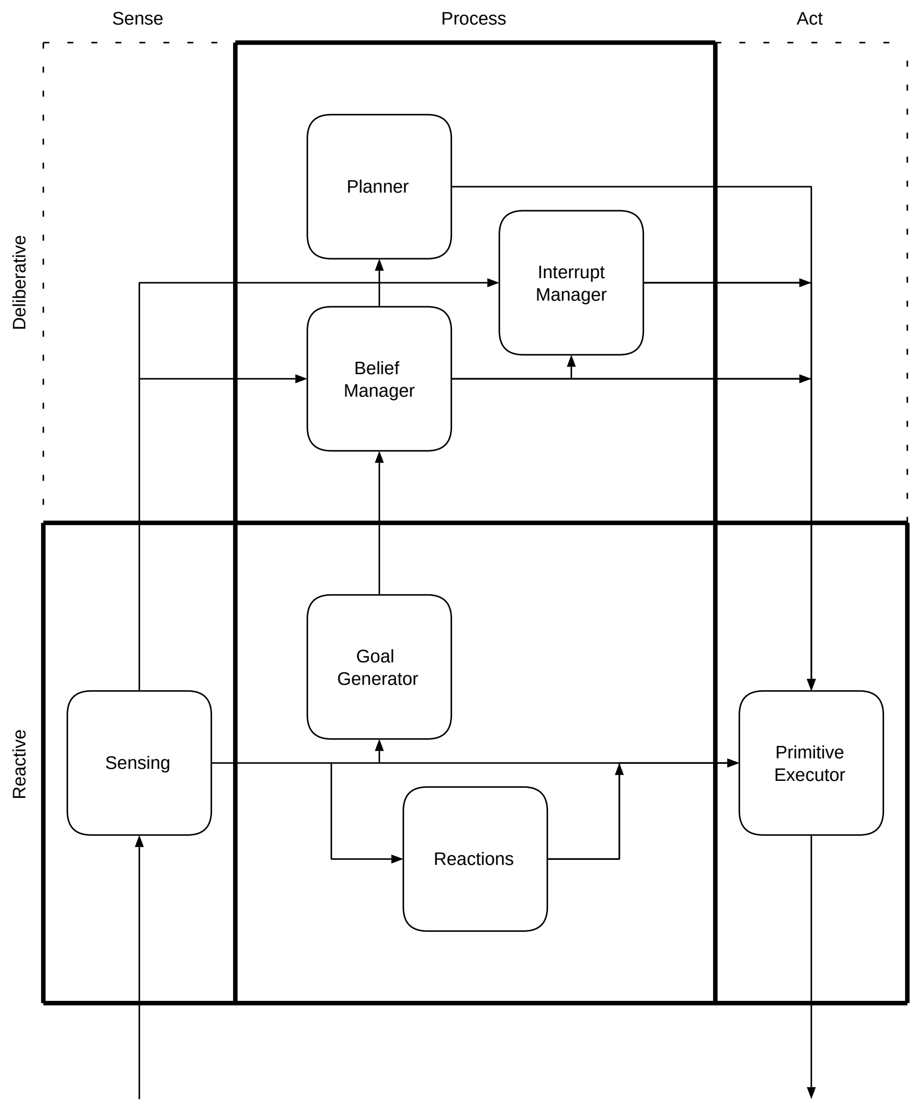

Figure 8.3: Agent Architecture Design For Iteration 2. The addition of the interrupt manager allows the agent to halt plan execution processes if they are no longer necessary.

Now that we have introduced the ability to interrupt plan execution, it is desirable to see what other conditions might warrant an interrupt being generated. As we have discussed previously, the state of the game world can change quickly and unexpectedly. This can lead to situations where the agent is executing a plan for a goal that is no longer valid, but no new goal of higher importance has been proposed. Using the Capture The Flag scenario as an example, we can envisage a situation in which the agent is executing a plan to intercept the opposition's flag carrier (a goal with an importance of five) when another member of its team attacks the flag carrier and returns the flag, or the opponent scores the flag (both events cause the flag to be returned to the agent's team's base). If this happens then the agent should adopt another goal, probably one to score a point. If this new goal is of a lower importance (as it probably will be in this example), then it won't generate an interrupt based on importance. To remedy the situation we must give the interrupt manager the ability to generate interrupts when a plan is being executed but its goal is no longer valid. To do this, not only must the interrupt manager cause execution to be halted (in the same manner as previously) but it must cause the belief manager to deselect the current goal. This is necessary because a valid new goal with lower importance cannot be selected whilst the belief manager has a goal of higher importance selected.

The interrupt manager plays a similar role in the agent architecture as the environment monitors do in the CPEF continual planning system [Myers, 1999] (reviewed in Section 3.2.3).

8.2.1.3 Primitive executor

The design of the primitive executor requires little alteration from the design used in the previous iteration. The only necessary addition is to ensure that the execution of a plan can be halted when an interrupt event occurs.

8.2.2 Example implementation

We can now examine the changes in the example implementation of the agent architecture made necessary by the changes to the design.

8.2.2.1 Belief manager

The implementation of the belief manager is altered so that goals are selected immediately, without waiting for the end of a planning or execution process. The implementation is also altered so that the belief manager deselects the currently selected goal if it is instructed to do so by the interrupt manager. This is done by changing a number of the conditions of the agent's goal selection rules.

8.2.2.2 Interrupt manager

The implementation of the interrupt manager must deal with two problems; how to determine when an interrupt should occur and how to inform other components that an interrupt is required. Informing the other components is the simpler task of these two, so we will deal with this first. When the interrupt manager determines that an interrupt should occur, it adds a fact into the agent's database instructing the primitive executor to halt the execution of a specific goal. The name of the goal is included in the instruction in case there is a delay between the generation of the interrupt and the interrupt occurring. If this occurred then there would be a danger of the wrong execution process being interrupted. Although a faster interrupt could be generated by the interrupt manager somehow preventing the primitive executor from running at all (e.g. by terminating its execution thread, then restarting it), this is not desirable because the primitive executor is used by other components in the architecture (e.g. the agent's reactions) and this brute force method of interruption may interfere with such operations.

The design of the interrupt manager specifies two conditions that must be evaluated in order to trigger an interrupt. The first of these conditions is whether a the belief manager has selected a goal with a greater importance than the one associated with the plan currently being executed. This condition is evaluated by simply retrieving the relevant information from the agent's database, then comparing the importance of the two goals. If the selected goal's importance is greater than the importance of the execution goal, then an interrupt fact is added to the database.

The second interrupt condition to be evaluated is whether the current plan is valid, regardless of what the current goal is. There are potentially two ways to implement this evaluation. The first is for the interrupt manager to examine each of the preconditions for each of the actions in the plan and check that they are still satisfied in the game world. This would provide the most accurate way of checking whether the plan is still executable. Unfortunately, such an approach is also computationally expensive. This is a problem because whilst a plan is being executed the interrupt manager must perform this evaluation continually in order to avoid lagging behind the events in the agent's environment. Such a computationally expensive action would dramatically slow the processing of the agent. The second way of performing this evaluation is to encode the reasoning that would be done by the first approach into a reactive rule set. This is what is done in the example implementation. The interrupt manager contains a set of rules that check a number of conditions in the environment when a plan is being executed for a particular goal. For example, if the agent is executing a plan to intercept a flag carrier, then the interrupt manager checks that the flag is still not at the agent's base, and if the agent is attempting to score a point, the interrupt manager checks whether any other agent has taken the flag already. This reactive approach is possible because of the limited set of goals that the agent can have plans for, and the small number of factors that are actually relevant to the completion of most of the plans (mainly object locations). By taking this reactive approach, the interrupt manager can perform evaluations quickly and frequently, allowing the agent to keep its behaviour up to date with its environment.

8.2.2.3 Primitive executor

The only addition needed to the implementation of the primitive executor is that it should halt once a particular signal is received. This is done by adding a rule that is run before the TRP system is executed. If an interrupt has occurred for the current plan execution process then the rule resets the TRP system (removing any temporary execution data) and exits from the primitive executor.

Quantity	Value	
Iteration	**2**	1
Avg. Planning Time	**209 secs**	138 secs
Avg. Wasted Planning Time	**27 secs**	37 secs
Avg. Execution Time	**93 secs**	153 secs
Avg. Wasted Execution Time	**0 secs**	95 secs
Avg. Percentage of Goals Selected	**22%**	24%

Table 8.4: Results from the second iteration of the design process.

8.2.3 Analysis of example implementation

We can now examine how the changes to the design have affected the real-time performance of the example agent. The results from the experiments run with the example agent implementation developed by the second iteration of the design process can be seen in Table 8.4. The current results are shown in bold next to the results from the previous iteration. The experimental data were gathered in the same manner as for the first iteration (see Section 8.1.3). The results show that the agent no longer wastes any time executing plans when the associated goals are no longer appropriate. As a result of this, the amount of time the agent spends executing plans has decreased, and consequently it has more time to spend planning. This is reflected by the increase in planning time. Although the amount of planning time has increased, these results show a decrease in the amount of planning time that is wasted by the agent. It is likely that this change isn't connected to the alterations made to the agent's design because none of the interfaces to the planner were modified. Along with this unexpected positive change, the results show an unexpected negative change. The percentage of goals selected has dropped below its value in the first iteration. With less time spent executing, it should be expected that this value would rise as the agent is able to complete more plan-act cycles and therefore present the belief manager with more opportunities to select goals. The number could have decreased because on average the individual planning processes in this round of experiments took longer than in the previous iteration, and this reduces the number of plan-act cycles that the agent can complete. This increase in planning time could either be unrelated to the design changes, or it could reflect the additional overhead introduced by the changes to the design (although this seems unlikely).

Although the design proposed in this second iteration of the design process improves upon the previous design in most respects, there still appears to be a significant amount of time being wasted by the agent during planning processes. This time wasting behaviour is not strictly due to the speed of the planner (although a faster planner would waste less time), but is due to the fact that unlike the execution process, the planner is not interruptible in this design. The next iteration of the design process will attempt to remedy this.

8.3 Iteration Three: Simple Planner Interrupts

At the end of the last design iteration, the fact that the agent's planning process cannot be interrupted was highlighted as the possible cause for the amount of planning time wasted in the example agent implementation. For the third iteration of the design process the focus will be on ensuring that the planner is interruptible.

8.3.1 Design

As with the previous iteration of the design process, only the design of the altered components will be discussed. The design of the belief manager does not have to be altered any further. It has already been designed to respond immediately to environmental changes. This behaviour is enough to allow interrupts to be generated by a separate component.

8.3.1.1 Interrupt manager

The previous iteration of the design process specified the conditions that should trigger an interrupt for an execution process. These conditions are directly applicable to planning processes. The first condition is when the belief manager has selected a goal that is more important than the one currently being planned for (executed for in the previous iteration). The second interrupt condition is when the necessary conditions are no longer present in the environment to warrant a goal being planned for (i.e. the current world state is sufficiently different from the initial planning state). As the previous design for the interrupt manager already specified that these conditions should be detected, only two more alterations are necessary to support interrupts during planning processes. The first alteration necessary is to ensure that the interrupt manager also evaluates the interrupt conditions during planning processes and not just during plan execution. The second alteration is to connect the interrupt manager to the planner so that the interrupt manager can inform it that an interrupt is necessary. The revised architecture design is presented in Figure 8.4.

8.3.1.2 Planner

As in the last design iteration, the component that is being interrupted requires only a single alteration. In this case all that is required is for the planner to halt execution when input is received from the interrupt manager. There is no need for the planner to perform any additional reasoning, but to just cleanly stop working on the problem.

8.3.2 Example implementation

8.3.2.1 Interrupt manager

The new additions to the implementation of the interrupt manager involve the communication between it and the planner, and its behaviour during the planning process. Communication between the planner and

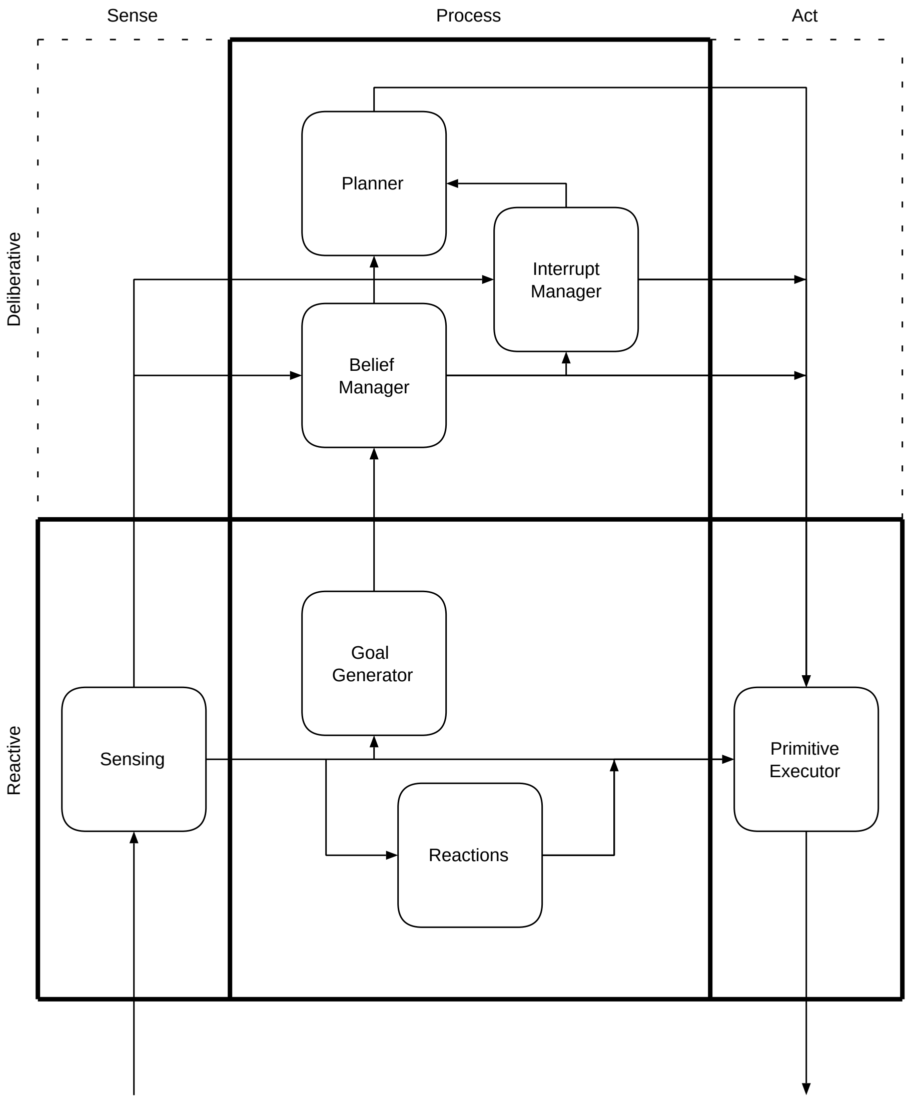

Figure 8.4: Agent Architecture Design For Iteration 3. The interrupt manager is now linked to the planner so that this can be interrupted if the current planning process becomes unnecessary.

Quantity	**Value**		
Iteration	**3**	2	1
Avg. Planning Time	**186 secs**	209 secs	138 secs
Avg. Wasted Planning Time	**0 secs**	27 secs	37 secs
Avg. Execution Time	**105 secs**	93 secs	153 secs
Avg. Wasted Execution Time	**0 secs**	0 secs	95 secs
Avg. Percentage of Goals Selected	**26%**	22%	24%

Table 8.5: Results from the third iteration of the design process.

the interrupt manager is implemented in the same manner as the communication between the interrupt manager and the primitive executor. When an interrupt event occurs, the interrupt manager adds a fact into the agent's database which is subsequently taken as input by the planner. The behaviour of the interrupt manager during planning is changed by adding conditions to its rules to allow it to monitor for potential interrupts during planning processes as well as during execution processes.

8.3.2.2 Planner

The sole necessary alteration to the implementation of the agent's planner is to ensure that it can be halted. As the agent's HTN planner runs in discrete cycles (one cycle represents one reduction or one attempt to satisfy a set of constraints), a flag could be added to the planner code to check for an interruption between cycles. This would be the simplest way of achieving interruptibility. Unfortunately, this simplistic approach does not provide the greatest amount of real-time control over the planner because there is no guarantee on the maximum amount of time the planner can spend in a planning cycle. Therefore, using this method would mean that the amount of time before an interruption event could have an effect could potentially be as long as a full planning cycle (which is an arbitrary amount of time). This type of approach (checking a flag set when an interrupt occurs) works for the primitive executor because every time the executor is run, this flag can be checked, and because the per-cycle execution time of the primitive executor is strictly limited. This is due to the rule-based nature of the primitive executor. The planner is not encoded in rules, but is implemented in standard Pop11 and run by the agent when a plan is required. To provide the necessary fine-grained control over the planner, it is executed in a separate processing thread which is allowed to run for a small amount of time every time the planner is required (much like the timing used to determine when the primitive executor checks its TRP-style rules). The rules that control when the planner is run check for the interrupt fact that is added to the agent's database by the interrupt manager. If an interrupt has occurred (i.e. if planning should be stopped so that the agent can do something else), the separate planning thread is just discarded. This allows the agent the same degree of control over the interruptibility of the planner as it has over the primitive executor. Simply discarding the planning process is safe because no other component in the agent uses the planner.

8.3.3 Analysis of example implementation

Experiments with the example implementation of the design proposed by the third iteration of the design process yielded the results shown in Table 8.5. The experimental data were gathered in the same manner as for the previous iterations (see Section 8.1.3). These results show that altering the design from iteration two to allow planning to be interrupted as well has had positive effects. The agent now wastes no planning or execution time, and the average amount of time an agent spends planning has dropped by a similar amount to the amount of time it was previously wasting during the planning process. The average amount of time the agent spends executing has also risen due to the reduction in planning time. This is a benefit because ideally the agent should be executing for as much time as possible, as during the planning process it doesn't provide much competition to the opposition (although its reactions allow it to execute the appropriate behaviour in critical situations). The complete eradication of wasted time during both planning and execution has also led to an increase in the percentage of goals that the agent can select. This is an expected result as the ability to interrupt both planning and acting when the previous goal is no longer valid provides a greater number of opportunities for the agent to adopt a new goal. This expectation aside, it is hard to be sure of the reliability of this figure, as the previous iteration of the design process caused this value to fall instead of producing the expected rise (although the new value for the percentage of goals selected is now higher than the comparable value in iteration one of the design process).

8.4 Iteration Four: Anytime Planning

The previous iterations of the design process have concentrated on identifying and removing the design flaws based on the amount of time wasted by the example implementation of the design. Now that the amount of time wasted has been reduced to zero, we can look for other, less obvious flaws in the agent design.

The results from the third iteration of the design process (in Table 8.5) show that the agent still spends a great deal of time planning. Although none of this time is "wasted" using the definition introduced in Section 8.1.3, it is still possible to argue that the agent is wasting time during both planning and execution. If a planning or execution process is interrupted and halted, then the time spent on that process is effectively wasted as the agent has not gained any direct benefit from the process (except the possible inadvertent gains made by taking up a potentially rewarding position through the interrupted execution of a plan), and the partial results of the process are discarded[9].

If the agent could anticipate when interrupts are likely to occur and use the results of partial processes, then the deleterious effect of this situation could be remedied. Unfortunately, it is not possible to take this approach to plan execution. Although the execution process can be interrupted at any time, at no point is there anything concrete that can be retrieved from this process to use at a later date (therefore not wasting

[9]The planner could make use of these partial results by building on them during subsequent planning processes. This is not done in order to fully demonstrate the power of anytime planning. See Sections 8.5 and 10.2 for further discussions of this.

the time spent executing before the interruption). Any gains made from a partially enacted execution process will be the result of actions taken during execution (e.g. gaining a favourable world position or observing an interesting event), not as the result of anything returned or stored by the execution process.

For the agent's planning process, this idea of using partial results is much more viable. Any processing effort expended during the planning process results in the space of possible plans being pruned to remove possibly invalid plans, so that the planning agent gains a better understanding of what a likely solution will be. If the planning process can be interrupted in sufficient time before an anticipated interruption, then this partial result can be used to guide execution. This approach will allow the agent to achieve its selected goals before they become invalid (i.e. before an interruption should occur) and reduce the amount of planning time spent on planning processes that are eventually discarded. This fourth iteration of the design process will investigate the changes that must be made to the design in order to support this behaviour.

8.4.1 Design

8.4.1.1 Planner

The design for A-UMCP, a planner that can be interrupted at any time and still return meaningful results, has been covered in detail in Chapter 5, and will not be reiterated here. Although many possible designs may exists for anytime planners, it will be assumed that the agent will be using A-UMCP.

8.4.1.2 Plan interpreter

We now require the agent to be able to execute plans that are not plans in the traditional sense. Instead, they are the results of partially executed planning processes that should be used to guide execution towards achieving the goal they were created for. As the functionality necessary to turn such results into actions does not exist in the current agent design, we need to add an additional component to the architecture; the *plan interpreter*. The new architecture can be seen in Figure 8.5. The plan interpreter is placed in the deliberative layer (although as we shall see, its design requirements inevitably lead to a reactive implementation) because it manipulates abstract information (partial plans) and is has functionality related to other components in the deliberative layer (e.g. the belief and interrupt managers).

As described previously, the proposed role for the plan interpreter is to take partial results from the planner and translate them into actions that can be performed by the agent's primitive executor. Although the exact design for the plan interpreter greatly depends on the precise nature of the planner being used by the agent, we can make a general assumption about such a mechanism. This assumption is the similar to the one used to develop the quality measure for A-UMCP (see Section 5.2), namely that the quality of the input to such an architectural component will be reflected in the quality of its output. If the partial result used as input to the plan interpreter lacks salient details of the intended behaviour, then the interpreter will have to guess at these details when it has to generate concrete actions for the agent to perform.

An important design requirement of the plan interpreter is that it is fast. This is important because of its role in allowing the agent to anticipate environmental events and take action before the moment has

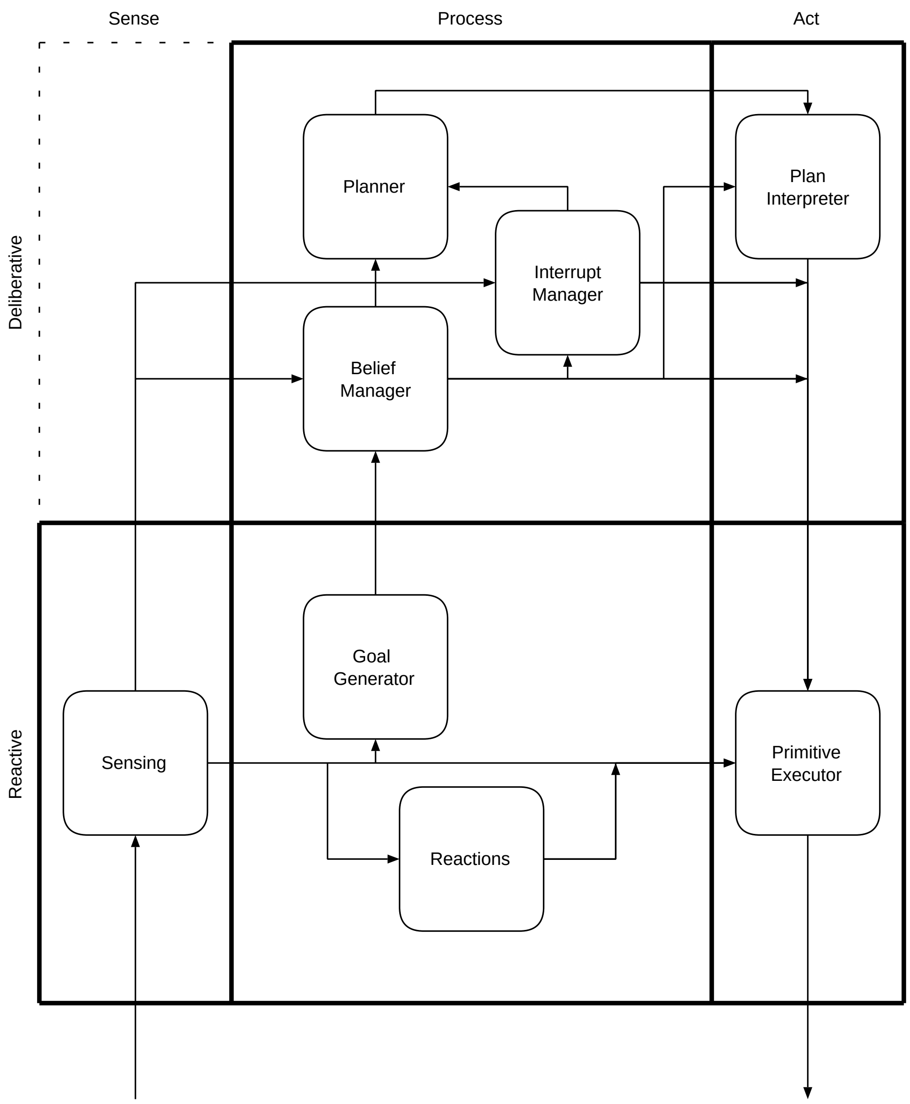

Figure 8.5: Agent Architecture Design For Iteration 4. The planner can now be interrupted at any time and this necessitates an additional plan interpretation step between the planner and the primitive executor.

passed. If the plan interpreter is too slow, then the agent will not gain an advantage by interrupting its planning process early (i.e. it may still run out of time before achieving its goal). If the time available to the agent was unlimited, then it could continue planning until a suitable solution has been found, but this is not always the case. This requirement for speed may necessarily limit the way the plan interpreter is implemented or may dictate the need for different methods of plan interpretation with differing processing profiles (cf. the different execution options used by the Phoenix agent [Howe et al., 1990]).

Because we are making the design assumption that the agent will be using A-UMCP, we can specify the design of the plan interpreter in slightly greater detail than if we weren't making this assumption. A partial result from A-UMCP will contain an arbitrary combination of primitive actions (i.e. those which can be passed directly to the primitive executor) and abstract tasks. The primary concern of the plan interpreter is translating these abstract tasks into primitive actions. Again we are faced with making a choice between a deliberative approach and a reactive approach. A deliberative approach would use reasoning techniques (possibly a domain specific planning approach) to interpret the abstract tasks based on the plan, the state of the world, and the agent's current state. A reactive approach would have the results of this style of reasoning encoded into rules that matched against the tasks and relevant states and then produced the appropriate interpretation. The reactive approach would be faster at run time, but the success of such an approach greatly depends on the number of possible abstract tasks and interpretations (as more abstract tasks would require more reactive rules to accurately encode all their possible reductions). A deliberative approach would be slower, but would be more generally applicable and robust in the face of unexpected inputs. Both the deliberative and reactive approaches could benefit by being augmented through learning. For example, learning commonly flawed reductions in the deliberative version, or learning direct mappings from abstract tasks to primitives in the reactive version (this could possibly be considered similar to humans learning reactions to particular abstractly defined situations). The choice of a suitable interpretation method is heavily domain and planner dependant, and as such will be left for the example implementation.

It is important to make a distinction between the proposed behaviour of the plan interpreter and the behaviour of the plan completion component of the anytime planner proposed in [Briggs and Cook, 1999] (reviewed in Section 4.4.2). The plan completion component in their anytime planner is intended to do the work of a planning process, albeit faster and in a more constrained manner (as these are the general properties of reactive planning). Because planning is involved, the plan completion component is still trying to solve a problem featuring a potential combinatorial explosion. In the agent architecture design, the agent's planner (A-UMCP) always has a complete representation of a plan, but it is specified in an abstract language. The plan interpreter must provide a mapping between this language and the primitive language used by the agent's primitive executor. Although the nature of this mapping will vary from domain to domain, it is intended that this mapping be of a fairly direct nature, allowing the problem to be solved in at most linear time. This is desirable because it prevents any significant delay existing between the interruption to the planner and a plan being returned.

8.4.1.3 Interrupt manager

The design of the interrupt manager must be extended to deal with a wider class of possible interrupts. This will enable it to spot more than just when the goal of a planning or execution process is no longer necessary. In this iteration, the interrupt manager must be able to spot situations in which, in order for the agent to avoid wasting planning time and to maximise its ability to achieve goals, it is necessary for the agent to interrupt its planner and use the partial results to guide execution. For example, if the agent knows that a plan for the goal it is currently planning for can typically be executed in ten seconds, then it knows that it must have at least ten seconds to execute the plan before the world changes to invalidate the goal. If the agent can anticipate when this world change is going to occur (e.g. through observing the regular behaviours of opposition agents) it can then trigger a planning interrupt at least ten seconds before the anticipated change.

Rather than being based on simplistic importance-based comparisons, this new type of interrupt is heavily reliant on time (as the above example demonstrates). Typically the interrupt manager must be aware of the amount of time left before the current goal becomes invalid and the amount of time it usually takes for the agent to execute a plan to achieve the current goal. The difference between these values represents the time available for the agent to plan in. The interrupt manager must examine these values to determine the appropriate (or even optimal, although this is a difficult judgement to make in such dynamic domains) time to interrupt the planner and start the interpretation and execution processes. Because of the dynamism of the domain and the absence of omniscience, it will be difficult for the interrupt manager to make consistently accurate decisions about when to interrupt, so it should involve either learnt probabilistic methods (although this may not guarantee success) or some value of assumed inaccuracy (e.g. five seconds either way, but this may cause the agent to waste time) when examining previously determined time values.

The interrupt manager is also the ideal place to use any performance profile data that the agent had about its planner. By examining such data, the interrupt manager can determine whether the planning process can be interrupted earlier than anticipated without any loss in quality, or whether it should be left longer in order to ensure a plan of a particular quality. In the first case, the agent should gain an advantage because it will have even longer to execute the plan before the anticipated change of goals occurs. In the second case, the risk of letting the planner run for longer than recommended by pure timing values may pay off with a plan that allows the agent to achieve its selected goal faster than had been anticipated, thus still allowing the agent to perform successfully even when planning for extra time. Alternatively, this second approach could result in the agent missing out on achieving its goal because it may spend too long planning and the resulting increase in plan quality is not enough to compensate for the lost time.

8.4.1.4 Belief manager

To support the new reasoning required by the interrupt manager, the belief manager must manage (i.e. generate, store and maintain) beliefs relating to time. More specifically, the belief manager must generate data on the times the agent typically takes to execute plans for particular goals, and how long the current goal is likely to be valid. Whereas the first of these values can be determined strictly by recording the times taken in particular instances, the second value (goal validity times) is more complicated. Values could be calculated simply by averaging out the timings of observed actions in the world (e.g. how long it usually takes for the opposition to score a point after the agent's flag has been taken), but they could also be calculated through less direct means. For example, if an agent is taking damage at a fairly steady rate, then the belief manager can estimate how much longer it will be before its energy reserves are exhausted. This estimate will represent the amount of time any healing goals will be valid for, because any attempt to heal after the agent is out of energy will result in failure (because the agent will be dead). The nature of how timing data is generated is another domain dependant feature, as it depends on exactly what goals the agent can achieve and what they depend on.

If the interrupt manager is to use performance profile data, then it is also necessary for the belief manager to generate and maintain these profiles. This can be done in a variety of ways depending on the type of planner being used, and the type of information required by the interrupt manager [Hansen and Zilberstein, 1996]. Although the details of constructing a performance profile are heavily domain dependant, we can present a general overview of the information that must be stored. A performance profile must provide the interrupt manager with a means of predicting the quality of a plan that will be returned from the anytime planner at a particular point in time. The information the predicted quality is based on can only be generated by monitoring the performance of the planner on a variety of problems. Depending on how the performance of the planner varies over the entire set of problems the agent faces, it may be necessary to construct a performance profile for each possible goal that the planner has to tackle.

8.4.2 Example implementation

8.4.2.1 Planner

The example agents uses the complete version of the A-UMCP anytime planner, the implementational details of which were presented in Chapter 5.

8.4.2.2 Plan interpreter

The possible design choices for the plan interpreter were presented in the design section. From these, the reactive approach has been chosen for the implementation of the plan interpreter. To implement this approach, the most general reduction for every abstract action is encoded into a set of rules. It is only possible to take this approach because of the limited number of abstract tasks used in the Capture The Flag domain. A degree of complexity in writing the rules was avoided by only considering a single abstract task at a time. The interpretation method could have been made more accurate if combinations of tasks had been addressed (this ultimately leads to reactive planning, an approach reviewed in Section 3.2.2), but the additional effort is not truly necessary to provide a working plan interpreter for the Capture

Partial Plan	**Necessary?**
`[[n1 scorepoint]]`	Yes
Interpretation	**Necessary?**
`[[n1 planpath Waypoint RedBase]`	No
`[n2 followpathnodes Waypoint RedBase]`	No
`[n3 pickup RedFlag RedBase]`	No
`[n4 planpath Waypoint BlueBase]`	No
`[n5 followpathnodes Waypoint BlueBase]`	Yes
`[n6 putdown RedFlag BlueBase]]`	Yes

Table 8.6: Example interpretation for an early plan.

Partial Plan	**Necessary?**
`[[n1 DO_NOTHING]`	Yes
`[n2 at BlueBase]`	Yes
`[n3 putdown RedFlag BlueBase]]`	Yes
Interpretation	**Necessary?**
`[[n1 planpath Waypoint BlueBase]`	No
`[n2 followpathnodes Waypoint BlueBase]`	Yes
`[n3 putdown RedFlag BlueBase]]`	Yes

Table 8.7: Example interpretation for a mid-process plan.

Partial Plan	**Necessary?**
`[[n1 DO_NOTHING]`	Yes
`[n2 DO_NOTHING]`	Yes
`[n3 followpathnodes Waypoint BlueBase]`	Yes
`[n4 putdown RedFlag BlueBase]]`	Yes
Interpretation	**Necessary?**
`[[n1 followpathnodes Waypoint BlueBase]`	Yes
`[n2 putdown RedFlag BlueBase]]`	Yes

Table 8.8: Example interpretation for a complete plan.

The Flag planning domain[10].

A typical rule in the example agent's plan interpreter translates an abstract task into its single most general (and therefore most widely applicable) reduction. These reductions are based on the methods presented in Appendix A. The simplicity of the Capture The Flag domain means that multiple interpretations for one task (dependent on context) can be avoided because general cases subsume less general ones, rather than being disjunctive (i.e. when only one of a set of particular reductions will be correct depending on the current state). For example, the interpretation of the abstract task to have an object not only assumes that the agent does not already have the object (as not doing so would greatly limit the applicability of the interpretation), but it also it assumes that the agent is not where the object is and must also plan a route between its current position and the object's position. How this choice of implementation for the plan interpreter affects the agent and how it relates to the agent's use of its anytime planner is examined in Section 9.2.

To illustrate how the plan interpreter works, and how its interpretations change as planning proceeds, we can use an example in which the agent is planning to score a point. In this example the agent currently has its opponent's flag and also already knows a path from its current position to its base. We will look at the interpretations of three possible partial plans for the `scorepoint` goal, each representing the results of a different amount of planning time. In the following plans we will use the variables `RedBase` and `BlueBase` to represent the positions of the opposition's base and agent's base respectively, `Waypoint` to represent the current position of the agent, and `RedFlag` to represent the opposition's flag. Table 8.6 shows the interpretation for the most abstract plan possible (one that consists of a single `scorepoint` task). The most general interpretation of this plan includes all of the actions necessary to score a point from scratch. Because the agent already has the flag and already knows the way back to its base, the first four actions are unnecessary. Table 8.7 shows the interpretation of a plan that has been reduced more than the first example. The reductions have removed any actions pertaining to the agent obtaining the flag and the plan just concerns itself with getting the agent back to its base (using the `[at BlueBase]` task). The most general interpretation of this task involves planning a path to the base then following that plan. Because the agent already knows a path to its base, the first action in the plan in unnecessary. Table 8.8 shows an interpretation of an almost complete task network. It is composed entirely of primitives so the interpretation just repeats these primitives. This is not really an interpretation as no abstract tasks are involved, but this situation may occur if the planner had yet to refine all of the constraints on the plan (therefore it would not be a traditional planning solution). Because all of the primitives in the plan are there for specific roles (as determined by the planner) the interpretation contains no unnecessary actions.

[10]It is interesting to see how planning effort, and therefore complexity, is shifted around the architecture. Writing a more complex plan interpreter would have reduced the need for extensive deliberative planning, and would have instead encoded the potential results of such a process in reactive form. When knowledge is provided in reactive form, it is the developer that is solving problems (rather than the planner), but complexity is still present due to the need to match rules against states.

Deficit	Execution Time (s)	Remaining Time (s)	Planning Time (s)
5	30	600	90
10	30	600	30
15	30	600	10

Table 8.9: Some examples of allowable planning times.

8.4.2.3 Interrupt manager

The addition to the interrupt manager of time-based interrupts is handled in the following way. To obtain the amount of time to spend planning, the interrupt manager subtracts the expected execution time for the current goal from the amount of time the current goal is expected to remain valid for. When this amount of planning time falls below a safety threshold (i.e. the largest amount of time the agent believes its timings can be wrong by) an interrupt event is generated to stop the planning process.

A good example of how the interrupt manager calculates interrupt times can be given with amount of time needed to score a point. To calculate this value, the agent needs to know how many points it needs to score, how much time is remaining, and how long it typically takes to execute a plan to score a point. This is then entered into the following equation;

$$p = \frac{r-(e*d)}{d}$$

In the equation p represents the planning time to be allowed, r is the remaining time in the game, e is the average execution time for a plan to score a point, and d is the amount of points the agent needs to score. This equation was used to generate the examples in Table 8.9 that illustrate how the agent can change its planning behaviour under time pressure.

The only time the behaviour of the interrupt managers differs from that described above is when the interrupt manager has performance profile data that is relevant to the current planning process. If the performance profile states that the interrupt manager can expect an increase in plan quality within a short period of time, then all attempts to generate an interrupt are postponed until after the time period has passed. For example, if the planner is currently offering a plan with a cost of five, but the performance profile predicts that this will be reduced to a plan with a cost of three in ten seconds, the interrupt manager can prevent interruptions for ten seconds (the cost threshold and lookahead are parameterised in the example implementation).

8.4.2.4 Belief manager

In the new design, the belief manager is responsible for generating the timing data used by the interrupt manager. As suggested in the design, it does this by timing all of its execution processes (to get data on its own behaviour) and by timing certain observable external events. The most crucial external event that the belief manager monitors is the time it takes for an opposition team member to score a point. It does this by starting a timer when its flag is stolen. If the flag is returned safely (i.e. no point is scored), then the timer is cancelled. If, on the other hand, the opposition score, then the time it took them is recorded and this is used to update an average value for this behaviour.

The belief manager is also charged with compiling performance profile data on the anytime planner for use by the interrupt manager. As the graphs of the planner's performance in Chapter 6 show, the performance of the planner can vary by a great deal even when tackling different problems in the same domain. Because of this, the agent requires an individual performance profile for each explicit goal it can tackle. It is possible to go even further than this and have performance profiles for particular groups of initial states when tackling specific planning problems. This would increase the fidelity of the performance profiles, but would come at the expense of more storage being required by the belief manager (especially if this approach was taken to the extreme of one performance profile per planning task).

In the example agent, a fairly simplistic approach is taken to the construction of performance profiles. The implementation of the belief manager defines a sample rate, and when planning occurs the quality of the plan is sampled at this rate. The data from this sampling are then indexed by time in the appropriate performance profile. If this is the first time the particular goal has been planned for, the information is simply stored by the profile. If the profile for the current goal has already been constructed then the new data is included in an average of all previous values for that time point. When a profile is constructed by averaging values from previous runs, it is important to bear in mind the monotonicity constraint on the planner. If averaging values from previous planning processes results in a performance profile that presents a non-monotonic relationship between plan quality and processing (e.g. when a planning process finishes earlier than another process for the same goal), then the non-monotonic parts of the performance profile are "smoothed" to present a monotonic relationship to the interrupt manager.

Compiling data in this manner creates performance profiles that represent *time-dependent utility functions* [Hansen and Zilberstein, 1996]. A more complex type of performance profile presented by [Hansen and Zilberstein, 1996] is a *dynamic performance profile*. Rather than basing the predicted quality of the partial result of the anytime process on purely time data, dynamic performance profiles allow the monitoring entity to judge the probability of getting a particular quality solution given the current quality of the current solution. This is a much more powerful solution than the one employed by the example implementation, and as such it requires more complex computation when building and using the performance profiles. Due to the relatively simple nature of the planning domain, such complexity was thought to be unnecessary in the example agent (as the overhead may have outweighed the potential benefits).

8.4.3 Analysis of example implementation

The example implementation of the design produced by the fourth iteration of the design process was put through the same experiments as the agents from previous iterations. The results from these experiments can be seen in Table 8.10. The experimental data were gathered in the same manner as for the previous iterations (see Section 8.1.3). The results show that by altering the agent design to allow preemptive

Quantity	**Value**			
Iteration	**4**	3	2	1
Avg. Planning Time	**157 secs**	186 secs	209 secs	138 secs
Avg. Wasted Planning Time	**0 secs**	0 secs	27 secs	37 secs
Avg. Execution Time	**135 secs**	105 secs	93 secs	153 secs
Avg. Wasted Execution Time	**0 secs**	0 secs	0 secs	95 secs
Avg. Percentage of Goals Selected	**32%**	26%	22%	24%

Table 8.10: Results from the fourth iteration of the design process.

interrupts of an anytime planner we have reduced the amount of time the agent spends planning. This has resulted in an almost identical increase in the amount of time the agent spends executing plans. This is important because it means that the agent is spending an increased amount of time acting out goal-directed behaviour, rather than relying on its reactions whilst planning. The new design has also resulted in an increase in the percentage of suggested goals that the agent selects. This was expected as the new agent agent design again effectively speeds up the time the agent takes to run through a plan-act cycle, so it therefore has more opportunities to select proposed goals.

8.5 Summary

What we have seen through the preceding sections is the development of an agent that can behave in an intelligent goal-directed manner whilst dealing with the unique problems presented by a real-time, dynamic domain. Each iteration of the design process highlighted specific problems with a previous design (starting from a basic design) and then attempted to alter the previous design to remedy the problems. By introducing interruptions to execution then to planning processes, iterations two and three prevented the example agent implementation from wasting time pursuing goals that were either superseded in importance or no longer valid. Iteration four took some initial steps towards proving the validity of anytime planning as a method for allowing agents to generate goal-directed behaviour whilst staying responsive to a real-time, dynamic environment.

The cycle of design iterations is potentially infinite, as the design could be improved in a variety of different ways depending on what the designer wanted to investigate and the problems and opportunities presented by the domain. Both the basic design for the agent and the direction taken by subsequent revisions of this design reflect that this is primarily a thesis concerned with how planning can be used successfully by an agent in a complex, real-time environment. Because of this, it may appear that there were obvious improvements that could have been made during the design process that were consistently ignored (e.g. interleaving planning and execution or replanning from a partial result). To avoid making unjustified claims about the success of the agent design, the next chapter takes a more detailed look at some of the key components of the architecture, and the effect they have on the agent's performance.

Chapter 9

Further Experiments and Analysis

This chapter analyses and evaluates the performance of the combined agent and planner designs. This is done by comparing an example implementation of the final agent design against agents based on designs from earlier design iterations (i.e. ones that do not make use of anytime processing).

9.1 Information Processing View

The agent architecture produced by the final design iteration in the previous chapter has some interesting features that set it apart from other existing architectures. The architecture has been designed as a hybrid architecture to take advantage of the strengths of both the reactive and deliberative styles of processing[1]. Although the example agent was designed in terms of distinct reactive and deliberative processing layers, when the agent is acting in its environment, the distinction between the layers becomes less well defined. Because the agent uses an anytime planner, it can alter the amount of time it spends deliberating. When it interrupts a planning process and interprets the result, the processing required to interpret the plan is performed reactively by the plan interpreter. Therefore, when solving a problem (i.e. trying to achieve a goal) the amount of deliberative and reactive processing is variable depending on the current state of the agent and its environment. At one extreme, if there is no pressure, the agent can be completely deliberative. This involves allowing a planning process to run to completion, then the primitive executor can simply execute the plan primitives. At the other extreme, when the agent is under a great deal of (time) pressure, it can immediately interrupt its planning process and pass the planner's initial guess to the (reactive) plan interpreter. The plan interpreter then generates a plan from this guess and passes it to the primitive executor. At this extreme the agent generates the behaviour to achieve its goal in an almost completely reactive manner. As these examples show, the anytime planner allows the agent to alter how reactive it is depending the pressure put upon it in its environment. This means that it can always behave in a goal-directed manner (due to the involvement of planner) whilst responding to its environment

[1] It is arguable that every component in the architecture must ultimately be implemented reactively, including the inner workings of the planner. This evaluation, and the previously presented design, distinguishes deliberative processing from reactive by its ultimate aims (what-if reasoning) and by its structured manipulation of abstract data.

with the appropriate speed. Also, this behaviour distinguishes the designed agent from the majority of agents reviewed in Chapter 7. The closest observed behaviour in the reviewed literature comes from the Excalibur agent (see Section 7.3.2) which uses an anytime planner to provide a "continuous transition from reaction to planning" [Nareyek, 2002]. The crucial difference is that the Excalibur agent does not have a reactive component to deal quickly with situations where time is strictly limited.

9.2 Plan Interpretation Results

This section investigates the performance and behaviour of the plan interpreter described in Sections 8.4.1.2 and 8.4.2.2. It is important to examine this component of the architecture because it is only through the plan interpreter that the effects of the anytime planner can be observed. The important aspect of its performance is what primitive plans it produces from the partial results provided by the anytime planner. It should be expected that the interpretation preserves the planner's monotonic relationship between processing time and result quality. If the quality of the primitive plans produced as output from the interpreter is independent of (or at least only barely influenced by) the quality of the abstract plans used as its input, then the agent has no incentive to plan for any length of time. We can examine two examples to highlight the problem this causes. In the first example we can imagine an interpreter that uniformly produces high quality plans no matter what input it receives. In this case the agent gains no benefit from planning at all, as any input to the interpreter will result in a high quality plan, so the most abstract plan available (the HTN root node) can be returned immediately. For another example we can imagine an interpreter that produces a plan of random quality no matter what input it receives. In this case the agent has no idea what length of time it should plan for, because it has no guarantees that producing a high quality abstract plan will result in a high quality primitive one after interpretation. To avoid making planning time either irrelevant or unpredictable, the output quality of the plan interpreter should be dependent on the quality of plans it is given to interpret, and this relationship should preferably be monotonic (to avoid making prediction difficult).

Because of the proposed nature of the plan interpreter (i.e. fast and computationally cheap), it is unlikely that this monotonic relationship will be perfectly preserved. There are a number of reasons for this. If, for example, the interpreter was based on some kind of learning algorithm, then it would be likely that it would produce better interpretations for the cases it had experience with as it would have more of a chance to learn accurate mappings between abstract actions and primitives. It would produce worse interpretations for those it was unfamiliar with because it would have had less chance to learn the appropriate interpretations. This would not necessarily provide the monotonic behaviour required. For another example, if the interpreter was based on a reactive system then the interpreter would produce better interpretations if the current state of the world was similar to the state of the world in which the agent's designer had anticipated plans being interpreted. It would produce worse interpretations if the state of the world was quite different from the world the designer had anticipated. This again would not necessarily provide a direct relationship between input and output quality. It is the reactive case that we are dealing with in the example agent, so this is the situation we are facing with respect to monotonicity.

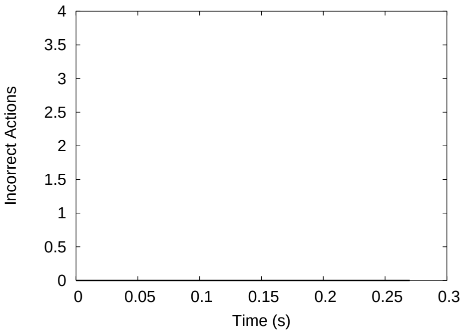

Figure 9.1: Interpreted plan quality when the current world state is identical the predicted most general state.

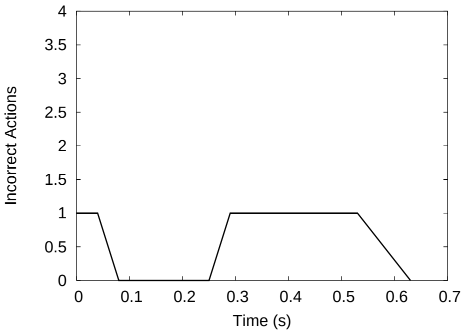

Figure 9.2: Interpreted plan quality when the current world state differs from the most general state by one literal.

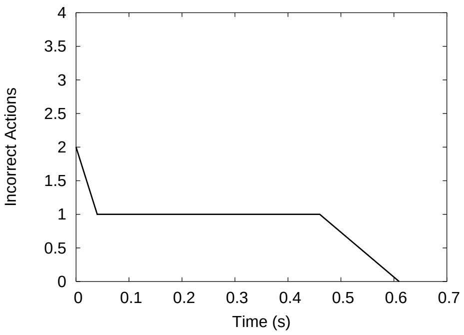

Figure 9.3: Interpreted plan quality when the current world state differs from the most general state by two literals.

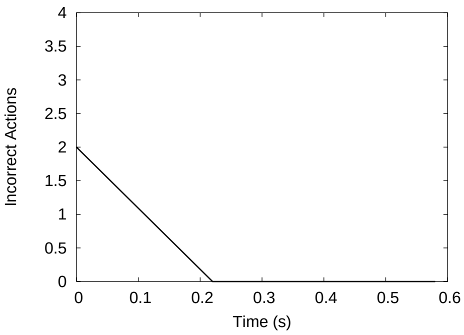

Figure 9.4: Interpreted plan quality in a different example in which the current world state differs from the most general state by two literals.

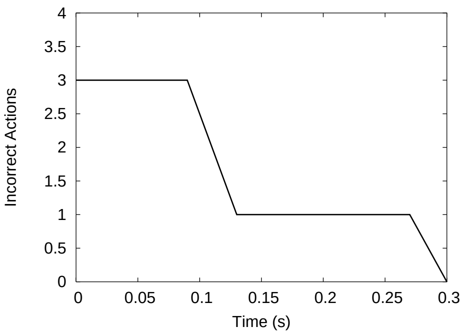

Figure 9.5: Interpreted plan quality when the current world state differs from the most general state by three literals. This represents the greatest possible difference in this example.

As has already been discussed in Section 8.4.2, the plan interpreter in the example agent implementation reactively translates abstract plans into the most general primitive plan possible. There is one specific world state that exactly matches the most general interpretation, so if this is the current state when the plan interpreter is used, then the translation between the partial result and the primitive plan will be flawless (i.e. the world is exactly as anticipated). There are many other states that match the general interpretation with varying degrees of accuracy (e.g. missing one or more of the assumed preconditions). If the current state is one of these states (i.e. not the exact match for the most general plan) then the interpretation will be flawed in some way. The example interpretations in Table 8.6, Table 8.7 and Table 8.8 demonstrate how flaws are introduced into the interpretation when the current state differs from the most general state.

Figures 9.1 to 9.5 demonstrate how the success of plan interpretation varies with the world state for the example agent planning to score a point. The data used to generate each graph were measured by examining the solution available from the planning process after every cycle, and working out how successfully it would be interpreted. The success of the interpretation process was measured by counting the number of actions included in the interpreted plan that are not present in the correct primitive plan for the given goal. This success measure (number of misplaced primitives) is indicated on the y-axis of the graphs. Planning time is indicated on the x-axis. All graphs end at zero difference because eventually the planner finds either plans that are completely primitive (and correct) or plans of a sufficient quality (i.e. a low amount of abstractness) that they can be correctly interpreted. The graphs end when the planning processes they are drawn from find the first solution. This is used as the stopping condition because there is no need to use the plan interpreter once a complete primitive plan has been found.

Each of these interpretation processes start off with a translation of the planner's HTN root node. This node is translated directly into the interpretation for the most general case. As the processes go forward in time, the planning process strives to make the partial results more like the actual environment. This results in the interpreter having to make fewer guesses during the interpretation process, therefore producing translations that are closer to the appropriate plan for the current state (not the most general case).

Figure 9.1 shows a case in which the agent's environment exactly matches the most general case envisaged by the designer. Because the match is exact, there are no spurious operators during the interpretation process. The other figures show how the success of the interpretation alters as the environment becomes decreasingly like the most general case. Figure 9.5 shows a case in which the agent's environment is completely different from the most general case. As such it starts with three misplaced operators (all of them). The misplaced operators are gradually removed as the planner creates a plan that fits closer to the current state. Eventually, when the planner finds a completely primitive plan, the interpretation shows no misplaced operators. The cases between these two extremes show that more misplaced operators appear in plan interpretations as the environment shows more differences from the most general state.

Figure 9.2 presents an example where the desired monotonic relationship between the quality of the input plan and the quality of the output interpretation is broken. In this case, the initial guess (which is wrong about one primitive) is quickly replaced by an input plan that can be interpreted correctly. Unfortunately, later in the planning process this abstract plan is replaced by one that re-introduces the error of one primitive. This is probably due to a plan being found with the same quality of the previous plan, but with some misplaced tasks that have yet to be removed. This is an unfortunate side-effect of the combination of the behaviours of the planner (it cannot prune some non-solutions without further reductions) and the plan interpreter.

There are a few more general observations that can be made about the performance of the plan interpreter. First, the interpreter can never produce plans that are better than the complete primitive plan that would eventually be produced by the planner, though interruption and interpretation can produce better plans at earlier stages in the planning process. This is reflected in Figures 9.1 and 9.2 by the indication of plans with no misplaced operators being available before the planning process is finished. Second, this interpretation method can produce plans that are wrong in two distinct ways. First, a plan can be wrong because it contains surplus operators in addition to those necessary to complete the plan. The potential success of such a plan varies depending on what the additional operators are. If they do not clobber any of the actions necessary to complete the plan (e.g. an additional `planpath` primitive in the example domain) the plan can still be successful. If they do clobber necessary actions (e.g. move the agent to the wrong place to pick up the flag) then the plan will not succeed (although this is also dependant on the orderings of the necessary and surplus steps). The second way of a plan being wrong is by not including an action necessary to achieve the goal. Such plans can never succeed as they lack a critical step to guide the agent. Using the most general interpretation rarely produces plans with this more serious flaw because "most general" is taken to mean that no assumptions are made about what the agent has already done. An invalid interpretation of this second type can only be produced using the most general interpretation approach if the partial result from the planner omits the necessary steps from the plan. This is a possibility if the partial result is about to undergo a constraint refinement step. For example, if the planner was constructing a plan to change the agent's position and was refining a plan that did not include the actual movement action (e.g. if it was being replaced by a `DO_NOTHING` action in A-UMCP) then the plan interpreter would not know to include a movement primitive in the interpreted plan. This would lead to a plan that would fail to achieve its goal of moving the agent. This situation is generally avoided in A-UMCP by using the planning quality measure as a critic that removes plans containing invalid `DO_NOTHING` primitives from the search space (see Section 6.5).

In conclusion, we can see that the plan interpreter component in the example agent performs as required. The monotonic relationship between processing time and plan quality is mostly preserved because as the planner produces partial plans that are better suited to the agent's current environment, the interpreter is able to interpret these results with more accuracy. The actual profile of this relationship depends on how close the state of the current environment is to the state anticipated by the designer of the plan interpreter. In practice, the majority of plans produced by the interpreter allow the agent to achieve its goals, but usually with a number of surplus actions being performed on the way to doing so. Only very rarely are completely invalid plans produced. When they are produced, it is usually because an interruption has occurred just as the planner is about to refine a task that is missing an `at` task. This results in a plan that misses one of its key actions (e.g. moving to the point where the goal can be achieved). As mentioned previously, this situation is generally avoided when using A-UMCP, except in specific cases when the planning heuristic cannot determine whether it is safe to prune a plan containing a `DO_NOTHING` from the search space.

9.3 Performance Profile Results

The design sections for the anytime planner and the agent architecture both discussed the usage of performance profiles to guide the agent when interrupting its planner. To investigate the effects of using a performance profile, two versions of the agent were implemented, one that used performance profiles, and one that didn't. Both agents had all of the other behaviours of the agent produced by design process in the previous chapter. The agent that did not use performance profiles just interrupted its planner when the interrupt manager determined it was necessary. The agent using performance profiles delayed interrupting its planner if the profile predicted that an increase in plan quality was due shortly. The intuition behind this approach is that the delay in processing should lead to a better plan that will more than make up for the delay in its execution.

To attempt to demonstrate the effect of using performance profiles, both agents were monitored as they went through numerous repetitions attempting to achieve a set of specific goals (this approach is explained in detail in the next section). This process resulted in the conclusion that behaviours of the two agent were indistinguishable. This is a rather disappointing conclusion, as the anytime algorithm literature claims that performance profiles are central to the abilities of anytime algorithms (e.g. [Zilberstein, 1996]).

The fact that using performance profiles has very little effect on the agent's behaviour can be put down to a particular facet of the agent's design and its implementation. This is that the planner usually takes a number of seconds to produce a new partial plan of a lower quality than the last one that was stored. This means that if the use of a performance profile extends the planning process by a second or two, this amount of time has very little effect on the resulting plan (often not even changing the measured quality of the returned plan).

This problem is caused because time has very different utilities for planning and execution processes in the example agent. As mentioned above, a couple of seconds is barely enough for the planner to increase the plan's quality above that of the last stored plan. Conversely, for execution processes, a couple of seconds can make the difference between success and failure (e.g. if the opposition manage to score a point whilst the agent is a few seconds behind their flag carrier). This difference in utility means that it is crucial for the agent to interrupt the planner as soon as possible in order to not get put at a disadvantage when executing the subsequent plan. Whereas the short extension (short to prevent the agent getting put at a disadvantage) allowed for planning time is not enough to make a significant difference to the plan. In addition to this, when small increases in plan quality are provided by performance profiles, they are rarely translated into significant changes in the plan that the agent executes. This is because these small changes are usually overwhelmed by the translation done by the plan interpreter. If the plan interpreter was more sensitive to the quality of its inputs (although this is probably not possible in the Capture The Flag planning domain), or the planner could increase the quality of plans faster, then performance profiles might have a greater influence on the behaviour of the agent[2].

The greatest potential area for performance profiles to have an effect in is when the planner is interrupted when the current plan is missing necessary actions to achieve the goal. This situation was previously discussed as the only time the interpreter can produce a flawed plan. There is no way for the planner to tell that it is dealing with such a plan (as this is what continuing the planning process is for), but if the performance profile introduced delays that caused the current flawed plan to be replaced, then this could alter the agent's behaviour. In the current agent, the chances of this happening are negligible.

In conclusion, it appears that there are a number of factors that have contributed to performance profiles having no noticeable effect on the behaviour of the example agent. From the anytime algorithm literature it is clear to see the effect they are intended to have, and that they are effective in a number of other domains. As such, this must be recognised as an area in which further study is needed. One area where such study may reap benefits is investigating the concept of a dynamic performance profile (as discussed in Section 8.4.2). Although, considering what was presented previously, it is possible that the additional power they provide may still have little effect in the example implementation.

9.4 Agent Behaviour Studies

Having previously looked at specific elements of the agent architecture, we will now investigate how anytime planning affects the overall behaviour of the agent. To allow us to focus on the aspects of behaviour primarily affected by the agent's planner, the agent will be analysed in a number of behavioural set-pieces. This allows us to clearly observe the effects of the anytime approach to planning, without these effects being overwhelmed by the other components of the architecture. After this more focused investigation, the anytime behaviour will be examined in context in a full game of Capture the Flag.

In the following studies a naming scheme will be used for the agents in the experiments. The term *A-agent* (anytime agent) will be used to refer to agents that are based on the design produced in the final iteration of the design process from Chapter 8. The term *N-agent* (non-anytime agent) will be used to refer to agents that are based on the basic design produced in the first iteration of the design process from Chapter 8. If the agent being discussed is in opposition to the agents being investigated, and not being investigated itself, its mnemonic will be prefixed with an *O* (opposition) (e.g. an *ON-agent* is a non-anytime agent placed in opposition to the agents in an experiment).

Agents from the other iterations of the design process are not considered in these experiments for two reasons. First, their novel (in terms of the previous designs) features are also included in the *A-agent*. Second, the abilities that define them (interruptible execution and planning processes) are not particularly complex and their advantages can be easily comprehended. As the ability for an agent to use an anytime planner (i.e. a planner that can be interrupted *and* return useful results) is novel and results in behaviour

[2]It is important to remember that the planner has been artificially slowed for the example agent (see Section 8.1.2.9). This could also be contributing the failure of performance profiles to make a significant impact on agent behaviour.

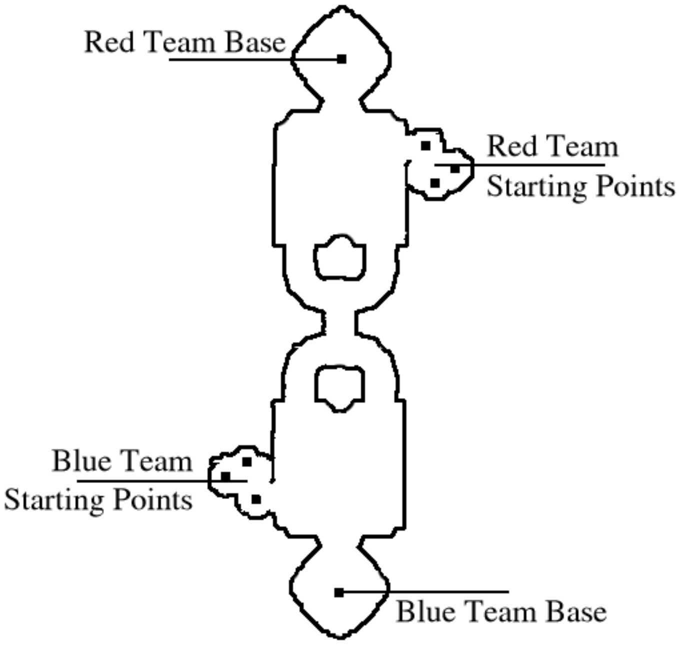

Figure 9.6: A map of the Capture The Flag experiment level.

that is less easily explained, this type of agent will be the focus of the experiments.

All of the experiments take place in the level mapped out by Figure 9.6. One team of agents plays the role of the blue team, and the agents in opposition to them play as the red team. These teams are named this way because of the colour of the lighting in their bases. These team names are not important and will not be referenced in the following experiments.

9.4.1 Study 1: Scoring under pressure

The anytime capabilities of the planner are only used when an *A-agent* is placed under time pressure. This first investigation will examine how using an anytime planner produces different behaviour as the agent is placed under varying amounts of pressure. To do this, we will concentrate on a specific game situation; the agent's team is a large number of points behind in a game of Capture The Flag, and it only has a limited amount of time to attempt to make up this deficit. To prevent any other behaviours influencing the agent's actions, we will start by investigating this situation when the agent is left unopposed and unassisted to attempt to score the necessary points. Although this is unrealistic in terms of gameplay, it allows us to produce more reliable results as they are less likely to be influenced by the actions of other agents. In the following experiments, the performances of two agents are examined; an *A-agent* and an *N-agent*. The game is set up to last for ten minutes, with a single agent attempting to score enough points to win the game.

Pressure is placed on the agent by varying the point deficit at the start of the game. The number of points required to win a game is a source of time pressure because the greater the deficit, the faster points need to be scored to make up the deficit before the game finishes. In the architecture, the belief manager keeps track of the current game score and the average time the agent takes to score a point. The interrupt manager uses this information to calculate how long the agent should plan for. Each time the planner is called, the interrupt manager lets the planner run for the maximum amount of time possible taking into account the number of additional planning and execution processes that must be executed in order to win the game. For example, if the agent needed to score three points to win the game, with three minutes remaining and point scoring execution processes taking thirty seconds each, the interrupt manager would allow the planner to run for a maximum of thirty seconds as that would mean that the three necessary planning and execution processes could be run before the end of the game (additional examples were presented in Section 8.4.2.3). The effect of the varying pressure on the agents is visible as the number of points the agents score. As the time pressure on the *A-agent* increases, it is able to increase the speed with which it scores points. It does this by interrupting its planning process earlier when the pressure is greater. As the time pressure on the *N-agent* increases, it is not able to increase the speed at which it can score points because it is not able to alter its planning time (and execution time is fixed by the environment).

Deficit	***N-agent* Score**	***A-agent* Score**
9	13	13
10	13	13
11	13	13
12	13	14
13	13	14
14	13	14
15	13	15
16	13	15
17	13	15
18	13	15

Table 9.1: Results from agents attempting to score under pressure.

The results from the experiments can be seen in Table 9.1. The first entry in the table shows experiments in which the agents are required to score nine points in order to win the game. Both agents manage this easily by each scoring thirteen points. Because both agents can easily achieve the implicit goal of winning the game (enough planning and execution processes can be executed to achieve the necessary score), neither agent is under pressure. This lack of pressure means that the *A-agent* does not use its anytime behaviours and therefore the architectures of the agents are functionally identical. From this initial experiment we can extrapolate a fact about the *N-agent*. Because it is not able to alter its planning or execution behaviour, thirteen points represents the maximum number of points it can ever score in the ten available minutes. This can be observed in the experiments as when the point deficit (and therefore time pressure) increases, the *N-agent* fails to increase the amount of points it scores.

As we have stated previously, the *A-agent is* able to change its behaviour. It can score more points as the pressure on it (i.e. the point deficit) increases. This can be seen as the numbers of points scored

	Average *ON-agent* Team Score	σ	**Average Own Team Score**	σ	**Average Deficit**	σ
N-agent	11.4	0.55	4.6	0.89	6.8	0.84
A-agent	11	0.71	6	1	5	0.71

Table 9.2: Resulting scores from teams of agents attempting to score under pressure.

changes with along with the point deficit in Table 9.1. If we look to the bottom of the results table we can see that the *A-agent* is not able to score more than fifteen points in ten minutes. To score fifteen points the agent must interrupt the majority of its planning processes very early. As the planner cannot be interrupted any earlier, to score any more than this the agent's execution time must be reduced, but this is not possible (the agent can only move at a fixed speed). This means that fifteen points is the maximum possible score for the *A-agent*. Using this maximum possible score as a reference point, we can see that the *A-agent* always manages to score enough points to win the game whenever possible (i.e. when the target is less than or equal to fifteen), except in one case. When the point deficit is fourteen, the *A-agent* only manages to score fourteen points in the ten minutes. This is a problem because we know that the *A-agent* is capable of scoring the fifteen points required to win the game. Failing to score fifteen points in this case shows that the agent's interrupt manager has misjudged the interrupt times for the planner. The interrupt manager should have behaved as if the *A-agent* was under more time pressure. Behaving as if the agent was under more time pressure would have led to the planner being interrupted earlier, enabling the agent to score points faster and therefore achieve the winning score. To produce this behaviour the *A-agent* must either revise the time data being stored in the belief manager or cause the interrupt manager to include a larger safety margin when interrupting the planner. If the belief manager underestimates the amount of time it takes for the agent to execute a plan, then the interrupt manager may unknowingly leave the planner running for too long, leaving the *A-agent* with not enough time to execute the resulting plan. Revising the timing data in the belief manager may lead to more current and accurate timings that prevent an underestimate. Increasing the safety margin the interrupt manager uses would cause it to increase the amount of "extra" time it allows for plans to be executed beyond that suggested by the timings held by the belief manager.

Now we have highlighted the fact that an agent using an anytime planner can vary its behaviour under pressure, we can see how this reaction to pressure affects the agent in a more general, though not full game, situation. The situation we will use for this is an extension of the point deficit situation above. Instead of there only being a single agent in the game, we will now move on to a situation where the game is a competition between two teams of two agents. The opposition team is comprised of two *ON-agents*, and they are given a ten point lead to put pressure on the team being investigated. The job of the *ON-agent* team is to prevent the team being investigated from scoring, and only to attempt to score themselves if they are not being threatened by the other team. The teams being investigated will, as in the previous experiments, attempt to win the game by making up the points deficit. We will examine the behaviour of two teams. The first team is made up of two *N-agents* and the second is comprised of two *A-agents*. The experiments were run twenty times.

The results from the experiments can be seen in Table 9.2. Neither team ever scores enough points to win the game, even though the target of ten points is well within the scoring potential of both teams of agents. The teams never reach their targets because of the opposition put up by the agents they are competing against. Even with this opposition, it is evident that the *A-agents* fare better than the *N-agents*. On average, the *A-agents* manage to score more points than the *N-agents*. A side effect of them attempting to score more points is that the *ON-agents* are put under more pressure, and as such have less time to attempt to score points. This is reflected in the average final points difference between the two teams. The *N-agents* are on average almost seven points behind the *ON-agents* at the end of the game, whereas the *A-agents* are almost two points better off with an average deficit of five points.

As the only effective difference between the two teams of agents that were used in the experiment was the use of an anytime planner, we can attribute the greater success of the anytime agents to their use of an anytime planner. When observing the games played by the *A-agents*, the effects of the anytime planner only start to be observable later in the game. The most prominent of these effects is that the *A-agents* make more attempts to score points. This behaviour is only evident later in the game because it is only then that the amount of time remaining in the game starts to be less than the amount of time required to score the necessary amount of points to win the game.

9.4.2 Study 2: Intercepting the enemy

The next piece of isolated agent behaviour that we will study is the situation where an agent must intercept an opponent who has stolen its team's flag. In a successful interception, the intercepting agent kills the opposing agent and picks up the flag. In a failed interception the opposing agent gets the flag back to its base (possibly killing the intercepting agent in the process). The agents both start at fixed points and restart from these points if killed. From its starting point, the opposing agent aims to get the agent's flag from the agent's base and return to its own base. If it succeeds then it will then try to do this again from the starting point of its base. If it fails (i.e. if it is killed by the agent) then it must try again from its original starting point. The agent being examined moves freely around the world and will start intercepting the opposing agent from whatever position it ended up at after its last attempt. This complements the previous study as it involves a greater level of interaction between opposing agents. This means that the agent has to tackle a greater level of dynamism and unpredictability. The scenario is set up with one *ON-agent* only attempting to score points (it will not generate any other goals) and the agent being analysed only attempting to prevent the opposition agent scoring (it will not generate any other goals either). Three different versions of this scenario will be run.

The first scenario has the *ON-agent* attempting to score points without an opponent (i.e. similar to the previous experiment). The second scenario features an *N-agent* as the intercepting agent. The third scenario features an *A-agent* as the intercepting agent. In this experiment, we are interested to see how well the agents being analysed manage to restrict the scoring of the *ON-agent*. The faster and more reliably the agent attempts to stop the *ON-agent*, the less points it should be able to score, so the final score reflects the performance of the agent being analysed.

	Average *ON-agent* Score	σ
Unopposed	13	0
N-agent	8.6	0.89
A-agent	5.8	0.84

Table 9.3: Results of flag carrier interception experiments.

The results from the experiments are presented in Table 9.3. The games were run twenty times for ten minutes each, and the average score of the *ON-agent* was recorded. When the *ON-agent* is unopposed it scores thirteen points. This is the same amount as the basic agent scores in the previous experiment. When the *N-agent* opposes the *ON-agent*, the *ON-agent* manages, on average, to score almost nine points. When the *A-agent* is in opposition, this figure is reduced to six. These results demonstrate that the *A-agent* is more effective at intercepting an opposition flag carrier. This success is due to the same reason that the *A-agent* was successful in the previous experiment. The *A-agent* can make more attempts to interrupt the flag carrier because it spends less time planning when under pressure. In this case, rather than coming under pressure towards the end of the game (as was seen the previous experiment), the *A-agent* is under pressure during every planning process because the opposing agent is always attempting to score a point. As the agent continually tracks the amount of time the opposition takes to score a point, it continually revises the length of time it can plan for. Once the *A-agent*'s flag has been stolen, it only plans for this amount of time before interrupting the anytime planner and executing the resulting (interpreted) plan. This enables a faster response from the agent (occasionally at the expense of a flawed plan), which then increases the probability that the *A-agent* can intercept the opposition agent before it scores a point.

9.4.3 Study 3: Full game

The final study of the behaviour of the example agent demonstrates how the anytime planner affects the overall performance of the *A-agent*, rather than its performance in selected scenarios. To do this, a team of *A-agents* is pitted against a team of *N-agents*. Based purely on the scores of the games they play, we can determine which of the two architectures is best for an agent playing Capture The Flag (although there are other quality metrics for their comparison). The experiments feature two different scenarios. The first is a team of two *A-agents* competing against two *N-agents* for ten minutes. The second scenario is similar, but with teams of three agents instead of two. Experiments with one agent per team were not carried out because the implementation of a number of the agent's components was performed with team play in mind. Because of this, when two agents play the game without teams they tend to perform badly and get stuck in deadlocks (when each waits for the other to make the first move) and repeating cycles of behaviour (one agent gains an advantage that it never loses through the timing of planning and acting processes). This does not happen with more than one agent per team because they do not all select the same goals at once (e.g. if one agent is carrying the flag, then the other agents will do something else as they cannot get the flag) and this variety of behaviour prevents standoffs between agents with goals

	Average Score	σ	Win %
N-agent	2.6	1.26	10
A-agent	5.1	1.2	90

Table 9.4: Results of 2 on 2 games.

	Average Score	σ	Win %
N-agent	2.4	1.17	30
A-agent	3.4	1.43	70

Table 9.5: Results of 3 on 3 games.

that cause deadlocks. Also, having more agents greatly increases the dynamism of the game, reducing the chances of agents getting stuck in repeating patterns of behaviour. Scenarios with more than three agents per team were not considered because, for convenience, the agents were designed and tested in (geographically) smaller levels that only support up to three agents per team.

The results from the two on two scenario can be seen in Table 9.4 and the three on three results can be seen in Table 9.5. In both scenarios the *A-agents* beat the *N-agents* in a significant percentage of the games played (ninety percent of the games in the two agents per team scenario and seventy percent in the three agents per team scenario). The *A-agents* are particularly dominant in the first scenario, on average outscoring their opposition almost two to one. Although the *A-agents* dominate the percentage of games won in the three agents per team scenario, their scoring rate isn't quite as prolific.

The increase in the number of agents on a team is directly related to the degradation of the performance of the *A-agent* team, or rather the increase in performance of the *N-agent* team. The principal strength of the team of *A-agents* is that they can react almost immediately to any important change of state in their environment. The *N-agents* can only react immediately if they are in a position (either physically or in their processing cycle) that allows them to do so (e.g. physically close to the focus of the subsequent goal or between planning and execution). When more agents are added to a team, the probability that an *N-agent* will be suitably positioned to respond to environmental change increases. For example, the more agents there are per team, the higher the chance that an agent will be near a member of the other team when that member captures the flag and the higher the chance that one or more agents will be at a suitable position in its processing cycle to adopt a new goal when necessary. This increase in the probability of being able to react appropriately makes the *N-agent* team more responsive to its environment, allowing it to compete more effectively with the team of *A-agents*. It is expected that this increase in reactivity will continue as more agents are added to the team of *N-agents*. If the geographical area in which the agents play was increased, then this would probably increase the advantage enjoyed by the *A-agents*. This hypothesis is based on the fact that the time it takes an agent to react to a particular occurrence is usually dependent on how far away the agent is from the focus of its behaviour (e.g. the opposition's flag), and a larger world would therefore increase the time it took for agents to act out plans. This increase in time would decrease the reactivity of the *N-agents* (who are tied to fixed planning times) and subsequently increase the advantage over them held by the *A-agents*.

9.5 The Iterative Approach to Design

Now we have seen how the agent design has developed, and have examined the end result of this design process, we are in a position to draw some conclusions about the design process itself.

The design process was based on an iterative methodology intended to explore the design space of possible agents for complex, dynamic worlds. This methodology was introduced in Section 2.1. At the start of Chapter 8 an initial design was created based on knowledge gained from the reviewed literature on agent design and planning in dynamic worlds. In many domains, deriving a design from such knowledge may have been enough to claim that the agent design was a good fit for its specified niche (as it had been designed to overcome some past mistakes). Unfortunately, due to the complexity of both game-like worlds and the hybrid agent design (which contains a number of interacting heterogeneous components), it would be difficult to make such a statement in this instance. Experiments with a proof-of-concept implementation of the agent design demonstrated that, although the agent could successfully perform the role it was designed for, a number of its components interacted in ways that introduced latency between the agent and its environment. Over a series of design iterations, this latency was minimised by altering and adding to the architecture. The alterations and additions were always guided by empirical data about the agent, ensuring that changes were made to areas of the agent design that required them (and not areas that were already functioning adequately).

The ability to empirically judge which areas of a design require improvement is the key strength of the iterative design approach used in this thesis. Not only can such an approach provide performance information on a particular component of a design (e.g. the agent's planner), but (critically for complex architectures) it can also provide information on how the interaction of various heterogeneous components affects the design (e.g. how the original interaction, or lack of interaction, between the agent's belief manager and planner contributed to wasted planning time in the agent implementation).

The advantages of an iterative design process do not end with the production of a final design well suited to its position in niche space. Such a design process provides the designer with a strong understanding of how the final design functions (i.e. how all its components interact to produce behaviour). This understanding comes not only from the empirical data generated during the design experiments, but through seeing the design change and evolve, and observing the effects of these changes throughout the functioning of the design. This understanding of the design gives the designer a greater insight into how to formulate experiments to test its overall behaviour. Understanding how a particular feature, or combination of features, contributes to the overall design also allows the designer to set up situations that isolate this set of features in order to demonstrate their behaviour. This was witnessed in this chapter when specific behaviours of the agent design were tested.

9.6 Summary

This chapter has demonstrated how the agent's plan interpreting component works, and how the assumptions made in its design affect its performance. In addition to this, the chapter presented the results of a number of experiments designed to show the effects using an anytime planner has on the behaviour on an agent. To examine these effects, the behaviour of agents using the A-UMCP anytime planner was compared to the behaviour of similar agents using an uninterruptible HTN planner. This comparison was carried out over a selection of scenarios isolated from full game of capture the flag in order to more clearly show the effects of the anytime planner (rather than the effects of other architectural components). The results of the experiments demonstrated how an agent using an anytime planner can change its behaviour when under time pressure. This ability allowed agents with anytime planners to outperform identical agents with non-anytime planners over a range of scenarios.

Part IV

Conclusion

Chapter 10

Contributions, Conclusions and Future Work

10.1 Contributions to Knowledge

The principal contributions made by this thesis were first outlined in Section 1.4 and will be reiterated here. This section also identifies a number of secondary contributions made by this thesis.

10.1.1 Principal contributions

This research has made two main contributions to knowledge. The first principal contribution is the Anytime Universal Method Composition Planner (A-UMCP), a domain-independent anytime planner. A-UMCP has defining features that set it apart from the small number of existing anytime planners. Whereas other anytime planners typically require a particular processing condition to be met before an interrupt can occur (e.g. an initial solution to be found, see Section 4.4 for more details on the this), A-UMCP can truly be interrupted at *any* time. A-UMCP is a completely *agent-oriented* planner, rather than being *solution-oriented* (see Section 3.2.1 for a discussion of these terms). This means that rather than creating exact solutions for problems, it generates information to guide (rather than completely dictate) the necessary agent behaviour. A side effect of this is that it is necessary for the agent to translate between the information generated by the planner, and the executable primitives of its execution system.

The second principal contribution made by this research is an agent architecture (shown in Figure 8.5) designed to support anytime planning and intelligent behaviour in real-time, dynamic, complex worlds (i.e. game-like worlds). These world features specified the position in *niche space* for the agent design. An initial naïve design was iteratively refined to produce the final architecture design. The changes made by each iteration were directed by empirical data from experiments with proof-of-concept implementations. This process explored the *design space* for agents in complex, dynamic worlds, until a design was found that fitted the specified niche. For any domain that has similar features to those that define game worlds, the final architecture design will provide intelligent behaviour whilst coping with environmental dynamism and complexity. The strengths of the architecture are its independent processing layers, and its separation of knowledge based on use, abstraction and rate of change. The weaknesses of the architecture include the necessity of an additional translation step between planning and acting that is not commonly present in other agent architectures (although without this additional step the agent would not be able to take advantage of the anytime features of its planner), a lack of explicit inter-agent communication features, and a lack of proactive intelligence. The strengths and weaknesses of the agent and anytime planner are explored in more detail in Section 10.2.

10.1.2 Secondary contributions

The development of the main contributions yielded a number of additional contributions to knowledge. These included;

- A heuristic that can be used to estimate how abstract a task in a recursively constructed hierarchical planning domain is. The heuristic is based upon the heuristic used in the HSP planner and works by treating an HTN task as a state-based planning problem. See Section 5.2.

- Two novel extensions to the algorithm used to calculate the HSP heuristic (see Section 5.3). These extensions are the memoization of the results of the heuristic, and the incremental calculation of the heuristic. Experiments in Section 6.4 demonstrated that memoization is very effective in speeding up the time taken to calculate the heuristic. The experiments also showed that the incremental heuristic is not as successful and can occasionally increase the speed of the calculation.

- A detailed and novel investigation into how planners can be made into true anytime algorithms. This examined previous attempts at creating anytime planning algorithms alongside the basic nature of planning and the tenets of anytime algorithms. The investigation found that previous attempts at creating planning anytime algorithms did not totally succeed because they either required an uninterruptible period of processing before they could be interrupted, or built plans action by action rather than working on the entire plan at once. See Sections 4.5 and 5.1.

- A proof-of-concept implementation of an agent architecture featuring advanced deliberative capabilities based upon the CogAff framework. Previous examples of CogAff research (including those reviewed in Section 7.2.1) have tended to involve mostly structured reactive processing.

- An example of an iterative design process being put into practice for the development of an agent architecture. This demonstrated how the design space for a particular niche can be explored and how empirical evidence can be used to guide the exploration of this space (see Chapter 8). This design process provided a deep understanding of the behaviour of the final agent design, which in turn facilitated the formulation of experiments to demonstrate the performance of particular features of the design (see Section 9.4). The strengths of the design approach used in Chapter 8 were discussed in Section 9.5.

10.2 Critical Evaluation of the Planner and Architecture

The architecture developed in Part III of this thesis, and the anytime planner developed in Part II, were designed to fulfil the aim of this thesis; to produce an agent that can behave intelligently in a world that is complex, real-time and dynamic. Most research inevitably falls short of its initial aims due to the need to focus on one specific area or another. It is important to examine how the final results of the research measure up against the initial aims, and to highlight any shortfalls.

Before these shortfalls are presented, it is worth reiterating where and how the research presented in the preceding chapters meets the aim of the thesis. In general terms, the aim of the thesis to produce an agent that can perform intelligently in a real-time, dynamic and complex world has been met. To examine this claim in more specific terms we can focus on the criteria used in the description of the aim. Intelligent behaviour is a very general concept, and as such its requirements are met in a number of ways. First, the use of a deliberative planner ensures that the agent is always acting in pursuit of its goals. Demonstrating flexible goal-directed behaviour is a certain sign of intelligence. Second, by being able to respond immediately to any change in its environment (by employing its reactions and an anytime planner), the agent acts intelligently by never getting left behind by unexpected environmental events. The overall intelligence of the agent comes from a combination of these features. One without the other would not be sufficient. It can also be argued that these features may become insufficient when the types of worlds inhabited by the agent start to allow more complex modes of interaction (although this may just require the further development of the agent's action components, not its entire reasoning system). Complexity in the agent's environment is overcome by ensuring that appropriate problem solving approaches are used to solve the different problems the agent faces (e.g. planning for future behaviour and reactive rules for behaviour execution), and by only using information of a particular granularity as input to these approaches. The agent deals with the problems presented by a real-time and dynamic environment by always having a response available to environmental change. Real-time responses are provided either by the agent's reactions, or by the agent interrupting its anytime planner in anticipation of the change, then executing behaviour to preempt the change (see Section 9.4.2 for an example of this in practice). This also contributes to the appearance of intelligence.

Although the aim of the thesis has been met in general, there are a number of specific areas where shortfalls can be identified.

It has been argued above, and previously in this thesis, that the agent appears intelligent because of its goal-directed and self-preserving actions. Because many games are heavily biased towards combat, players tend to judge an agent's intelligence on how it performs in combat at least as much as they judge it on long-term goal-oriented behaviour and self-preserving actions (cf. the studies on the Soar Quakebot in Section 7.3.1). Combat in the anytime agent produced in this thesis is controlled by the agent's reactions and primitive executor. Reactive combat behaviour was considered a secondary problem to long-term behaviour planning. This approach was taken because many other authors have carried out research into the reactive control of agents. To make the agent truly appear intelligent, more detailed work is necessary to dictate the (possibly domain-specific) nature of the agent's reactive behaviours in both the agent's reactions and primitive executor.

A less obvious problem with the agent's appearance of intelligence is in what causes it to display its "intelligent behaviour". The rather singular focus on being able to respond intelligently to environmental events has led to the production of an agent that only really does interesting things when under pressure. If the agent is not under pressure then it steadily follows a rather traditional sense-plan-act cycle. If the agent is to be considered to truly appear intelligent then it must have some more *proactive* intelligent behaviours such as anticipation (see Section 7.2.2) or explicit cooperation with other agents on its team.

When examining the internal processing of the agent, it is clear that although a lot of effort has gone into ensuring that little time is wasted when plans are being developed, the way in which completed plans are used provides more opportunities to further improve the agent's real-time performance. Currently every plan is used only once, but the nature of the divide between reasoning and execution knowledge (see Section 8.1.2.8) means that plans could be reliably reused if the environmental state contained the right facts for the plan. Plan reuse would then reduce the number of planning processes started by the agent, and therefore reduce the overall planning time. This simple and effective addition was not made to the agent in order to better highlight the effects of alterations made to the planning algorithm (e.g. making it interruptible). If fewer planning processes were started, the impact of these alterations would be lessened (although still noticeable and positive). A further alteration to the agent's internal processing that would reduce planning time would be to interleave planning and execution. This would involve the agent producing the first part of a plan and then executing this part whilst constructing the rest of the plan (which in turn is partially finished then executed). This is a change that would reduce the amount of time spent not executing (possibly making it appear more proactive) and would also reduce the observable effects of using an anytime planner. The combination of interleaving planning and execution with anytime planning could produce some very interesting effects. The planning process for each plan segment could be subject to constraints similar to those applied to the entire planning process in the example agent (e.g. planning must be completed before a particular time, for instance by the time the initial part of plan has been executed). This would lead to the agent making the most of the available planning time for each plan segment, rather than making the most of the available time for the entire planning process. Planning and executing in this manner would result in many small time savings at the expense of fewer greater savings (from complete planning processes). This could also possibly produce a greater overall time saving as the agent would be monitoring its processing in finer detail.

Another shortcoming of the research on the agent architecture is that although mechanisms are in place to deal with complex environments, these mechanisms are not thoroughly tested. The Capture The Flag domain in which the agent was tested is quite a simple planning domain. Also, the world of *Unreal Tournament* has so few modes of interaction (mostly shooting and moving) that no aspect of the agent was really faced with the complexity taken into account when it was designed. The agent certainly has the capabilities to deal with more complex environments, so in this respect the main aim of the thesis has been met, but its abilities have yet to be proven experimentally.

10.3 Comparisons To Existing Work

In Chapters 2 and 3 a number of problems with traditional planning approaches were considered. These problems included such things as the real-world (or virtual worlds) having durative actions whilst planning modelled these actions as having instantaneous effects, traditional planning not providing structures to represent many actions (e.g. loops and conditionals) and the possibility of the initial assumptions used by the planner becoming invalid in the execution environment. To overcome such problems, an approach was taken that was similar to the methods promoted by the field of continual planning (see Section 3.2.3). Rather than consider all of these issues in the design of a planning algorithm (as this would ultimately lead to something much more complex than a planner), the planner is situated in mechanisms that manage these issues, leaving the planner free to plan in a fairly traditional manner. In this case, the mechanisms surrounding the planner make up an agent architecture. The architecture handles many of the problems faced by the planner. For example, structured actions are handled by having action primitives executed by a reactive primitive executor (which has no such constraints on its action representation). The primitive executor also deals with durative actions (although this does not allow the planner to reason about time). The belief manager (Section 8.1.2.8) and interrupt manager (Section 8.2.1.2) in the agent architecture deal with the possibility of invalid planning assumptions by checking these assumptions before a plan is used, and monitoring the agent's environment for plan-affecting events

As this approach to planning is similar to continual planning (albeit with an additional leaning towards agent behaviour), it is interesting to see how the completed planner and agent combination satisfy the necessary features of a planning system that is intended to operate in the real world. These features were presented by [Pollack and Horty, 1999] and included in Section 3.2.3. The features will be listed here, along with the way in which the final design provides them (if it provides them at all).

- **Plan generation**: The A-UMCP planner is used to generate novel plans.
- **Commitment management**: The belief manager selects goals based on their associated importance measure. Only the most important goal proposed is acted on, and is immediately changed if a more important goal is selected or it becomes invalid.
- **Environment monitoring**: The agent's belief manager checks the requirements of a plan in the environment before it is executed. The interrupt manager monitors the environment for important changes during planning and execution processes.
- **Alternative assessment**: Whilst this is not explicitly dealt with in the agent, managing goals based on importance is a simplified approach to choosing between conflicting courses of action (i.e. the agent just picks the most important one).
- **Plan elaboration**: The planner aims to plan to its interface level. Once plans are complete they are elaborated by the agent's primitive executor to include up-to-date execution information.

- **Metalevel control**: The interrupt manager determines how much time should be spent on a particular planning problem based on the current state, the amount of time spent planning, and the estimated amount of time it will take to execute the plan.

- **Coordination with other agents**: Whilst this is not explicitly dealt with by the agent design, the example agent implementation makes use of limited communication with its team mates to inform them of its current state (principally whether it has captured the flag or not).

In Chapter 7 we saw a number of approaches to developing agents that had similar aims to this thesis. It is interesting to see what comparisons can be made between the reviewed approaches and the finished agent.

In Section 7.3 we saw two other academic approaches to creating agents for games; the Soar Quakebot and the Excalibur agent. These projects took notably different approaches to solving a similar problem, and whilst the agent developed by this thesis is different again, it does have some similarities to both approaches. More specifically, the work in this thesis combines the strengths of these approaches (the reactivity of the Soar Quakebot and the advanced planning behaviour of the Excalibur agent) to create an agent that is not tied to one style of processing. Although the individual components in the example agent may seem underdeveloped compared with the similarly intentioned components in the Soarbot and Excalibur agents (e.g. the reactive behaviours in the example agent fail to capture the level of detail encoded in the Quakebot's seven-hundred and sixteen behaviour rules), the architecture is applicable to more domains than either of these approaches.

The agent architecture and the derived proof-of-concept implementation are more directly comparable to previous work on hybrid agents. The work in this thesis draws heavily on concepts previously associated with a number of other agents and architectures. These concepts include things such as concurrently active processing layers (see the description of the TouringMachine architecture in Section 7.2.2, for example), solving particular problems with suitable processing approaches (for example, the way the SOMASS system, described in Section 7.2.6, divides up its problem solving) and representing knowledge in a way that reflects how it changes and what it is to be used for (see the work of Wood described in Section 3.2.4).

In Section 7.2.3 we saw a number of criteria that the authors behind the Guardian project stated are desirable for an agent to be successful in a complex environment. These criteria will be reiterated here along with ways in which they are satisfied by the example agent.

- **Communications**: The components of the agent architecture share data via a central database but selectively access data based on their processing styles and needs.

- **Asynchrony**: The agent can plan and act independently of events in its environment.

- **Selectivity**: Perception of environmental events is filtered by using different representations for the varieties of knowledge used by the different processing layers. Reasoning is selective based on the current goal of the agent (which itself is selectively adopted).

- **Recency**: This is not directly covered by the design of the agent, although old sense information is deleted from its database.
- **Coherence**: The design of the agent is focused around achieving a single explicit goal at a time, whilst enforcing its implicit goals (e.g. survival).
- **Flexibility**: The agent's reactions are specifically used to deal with unexpected environmental events. The anytime planner also contributes to the agent's flexibility as planning times can be altered in the face of anticipated environmental change.
- **Responsiveness**: Anytime planning allows the agent to adapt its behaviour when under pressure.
- **Timeliness**: When time constraints can be predicted during planning, the agent endeavours to respond before they are violated (by interrupting its anytime planner). When events are unpredicted, fast reactive rules are used to generate a response as fast as possible.
- **Robustness**: The anytime planner allows the agent to behave intelligently under resource constraints by running the planner for short bursts and by making the most of the limited time available to it.
- **Scalability**: The agent architecture is designed to scale to more complex problems by decomposing and representing problems in ways that minimise complexity (e.g. using reactive behaviour for movement and close control, and the planner for solving stable behaviour generation problems).
- **Development**: This is not something directly covered by the design of the agent, although the run-time compilation and use of performance profiles with the anytime planner would provide this ability (something that was tried but had little effect in the example agent, see Section 9.3).

The final comparison for this research is to the body of work produced by the Cognition and Affect group at the University of Birmingham. Part III of this thesis was situated within their framework for developing agent architectures. The results of the agent development show the benefits of the design stance promoted by the CogAff group. Also, the ability of the agent to act successfully in real-time comes as a direct result of having distinct layers of processing (each with an appropriate style of knowledge representation), an approach dictated by the CogAff architecture schema. The work in this thesis adds no new theoretical results to the work of the CogAff group, but can be considered an example of their principles put into practice on a substantial research project.

10.4 Wider Applications Of The Research

The research presented in this thesis is applicable in a wider context beyond computer games. The findings of the research are relevant in any domain where real-time goal-oriented behaviour is required from an agent, particularly where agents are placed under resource pressure (particularly time) and have the potential to predict when they may lose the opportunity to execute a plan due to environmental change. Domains where the knowledge generated by this thesis may also be useful include Robocup Rescue [Kitano et al., 1999], military simulations (e.g. [Atkin et al., 1999] and [Jones et al., 1998]), and interactive storytelling (e.g. [Cavazza et al., 2000]). Interactive storytelling is a particularly interesting domain for applying the agent research as it has the potential for complex planning problems relating to future story arcs, but it also requires some levels of real-time response to maintain interactivity.

10.5 Future Work

There are a number of directions that extensions to this work could take. Some of these extensions would help overcome some of the previously identified shortcomings of the research, others would add to the behavioural capabilities of the agent.

For the A-UMCP planner, the most necessary piece of future work is to correct the aforementioned problem (Section 6.6) with the heuristic miscalculating the cost of action tasks (g_{action}). The most natural way to overcome this would be to calculate values for such tasks after every cycle of the planner, rather than during the first cycle of the planner (or before run time). The costs of other tasks in the planner are calculated regularly, so this addition would not greatly alter the algorithm for the planner. One worry is that this alteration may deleteriously affect the real-time performance of the planner. The effect would be greatest in domains that are encoded with numerous action tasks that contain goal tasks as possible reductions (as this is the cause of the original problem and what would lead to further calculations in each cycle). Regardless of a possible reduction in real-time performance, this change should be made to allow the heuristic to perform better over all possible planning domains.

A more general direction for future work concerning A-UMCP is to further explore the planning behaviour determined by the heuristic it uses. An example of this would be to explore whether it would be possible to combine the current heuristic with a heuristic that actually drives the planning process towards solutions, rather than just towards lower cost task networks. Combining two heuristics in this way could allow the planner to trade off the speed with which it finds a solution against the speed at which it decreases plan cost (although there may be some overlap between these two aims).

One of the most necessary extensions for the agent research is to further evaluate the architecture by implementing the final agent design in a more complex domain. The planner was tested in a variety of domains (see the results given in Chapter 6), but due to the amount of work involved in developing a capable agent that fully embodied the design, a proof-of-concept implementation was only developed for one (relatively simple) domain. Games that would provide a greater level of complexity without requiring too great an alteration to the agent implementation include tactical or more “realistic” action games such as *Counter-Strike* [Valve, 2000]. To go beyond this in complexity would probably involve implementing the agent in a completely different style of game (e.g. a real-time strategy game such as *Warcraft III* [Blizzard, 2002]) or implementing the agent as an avatar to take the place of the human player in a strategic or tactical action game (e.g. *Metal Gear Solid 2* [Konami, 2001]). Implementing and testing the agent in a much more complicated domain would give a better idea of how well the agent architecture scaled to other domains, how well the design of the agent actually combats complexity, and therefore how well the design of the architecture fits its niche. It would also be interesting to see if the ideas presented by this thesis could be transferred successfully to physical robots where they would be faced with the complexity and dynamism of the real world.

A related piece of possible future work would be to evaluate how the architecture performs under much more restricted operating conditions. This would be aimed at investigating how the agent would fare in a real computer game. Restricting the operating conditions for the agent would involve limiting the amount of CPU time available to the agent, and also the amount of memory it was allowed for operation. Doing this would negate the need to artificially slow the anytime planner (as was discussed in Section 8.1.2.9). As the only processing intensive part of the agent architecture is the anytime planner, it is to be expected that it would be its real-time performance that would suffer the most. This decrease in performance should be mitigated by the planner's anytime capabilities as it will always make the most of the available processing time, even if this processing time is severely limited.

To continue the evaluation of the contribution made by the anytime planner to the behaviour of the agent developed in Chapter 8, the experiments in Section 9.4 should be extended to include comparisons between the *A*- and *N-agents* and a purely reactive agent. The reactive agent should be created by causing the anytime planner to interrupt every planning process immediately, resulting in the details of the subsequent behaviour being entirely specified by the (reactively implemented) plan interpreter. The aim of this extension to the experiments would be to determine whether the *A-agent* is more successful than the *N-agent* purely because it does less processing (is more reactive) than the *N-agent*, or because of the combination of the goal-directed and reactive behaviour (from the planner and plan interpreter respectively). The aforementioned simplicity of the Capture The Flag domain, combined with the speed at which the game is played, makes it possible that in the implemented agent, the reduction in processing plays a greater part in the success of the *A-agent* than the addition of goal-directed behaviour to its reactive processing. In more complex domains there will be much greater scope for the planner to have a more beneficial effect, as purely reactive processing will be less able to consistently achieve goals.

One area where the implementation of the example agent falls noticeably short of the agent design (although not short of the overall aim of the thesis) is in the way it handles the interpretation of abstract plans. The initial design described a hypothetical approach in which output quality was heavily dependent on input quality. This would lead to an agent being able to execute much better planned behaviour when it could dedicate a long time to planning, and worse behaviour after a reduced planning time. The way the interpretation process was implemented (see Section 8.4.2) provided this relationship between input and output quality, but provided only a small amount of difference between even the most extreme cases (see Section 9.2). This meant that the behaviour resulting from an early interruption was not much worse than the behaviour resulting from a late one. There may be many different ways to re-implement the interpretation mechanism (the agent's plan interpreter component), but one of the most interesting approaches would be to use case-based reasoning (e.g. [Aamodt and Plaza, 1994]). This particular technique learns mappings from situations (cases) to things that follow from these situations. In the instance of the interpretation component, it would have to learn the mappings from a combination of abstract plan tasks, goals, and critical aspects of the current state, to a suitable reduction (i.e. an ordered set of primitives). It could then use these mappings to interpret a given abstract plan into an executable primitive plan. This may provide the desired relationship between abstract plan input quality and resulting primitive plan output quality (although the success of this method would greatly depend on the number of cases the component had to learn from). Another possibility for the extension of the plan interpreter module would be to allow the agent to delay interpreting parts of a plan until it was time to execute them. With such an ability, the agent could interpret the start of the plan (e.g. the first abstract task) and then start executing it. Whilst it was executing this task, it could then start interpreting the next action to be executed, and so on until the plan is complete. This would have the advantage of giving the agent more time to perform interpretations for tasks later in a plan, and would allow the agent to gather more information from its environment to make the interpretation process more accurate.

Another direction in which the research could be extended is expanding the agent's architecture into the unused segments of the CogAff architecture schema (see Section 7.2.1). More specifically, adding a meta-management layer could add interesting behavioural capabilities to the agent. The meta-management layer could monitor the agent's balance between deliberation and reactivity and then alter the behaviour of one or more architectural components. For example, if the meta-management layer judged that the agent was never producing plans of a high enough quality, it could force the agent's interrupt manager to delay interrupts by an additional period of time and then observe the effects of this alteration. Conversely, if the meta-management layer judged that too many plans were not getting completed due to environmental change, it could force the interrupt manager to interrupt the planning process earlier than it usually would. Other possible uses of the meta-management layer include altering the way in which goals are accepted for execution (i.e. when an importance measure does not accurately model the world) (cf. [Wright, 1997, Section 5.3]) and preventing usually necessary reactions from overriding planned for behaviour (e.g. preventing the agent moving to safety when it is acting to prevent danger coming to another agent).

Future work is also necessary to determine the correct way to compile and use performance profiles for the agent's anytime planner. Although their use seemed redundant in the example implementation (see Section 9.3), there may be important reasons for this. A first possibility is that a combination of a number of features of the agent and its world (e.g. the interpretation method used and the planning domain) prevent the use of performance profiles from having a noticeable effect. Further possibilities include the idea the data in the performance profiles did not accurately represent the performance of the planning algorithm, and that the agent did not employ the data in such a way that the effects made a large enough difference to the result of the planning process. It is likely (given some of the other shortcomings of the agent implementation) that the first of the proposed faults played a large role in the ineffectiveness of performance profiles, but the other options cannot be ruled out without further development and experimentation. A first step in any future research of this nature would be to investigate whether the work on dynamic performance profiles from [Hansen and Zilberstein, 1996] could be successfully integrated into the agent design.

One aspect of the performance of the example agent that was not investigated is how well it performs as an agent for interactive entertainment. It is desirable to evaluate this because the agent has ultimately been designed for a computer game role where it would be interacting with (and possibly competing against) other agents (including human players). One key aspect of the agent's performance that determines how it will succeed as an entertainment agent is how believable (i.e. non-artificial) it appears to human players. Believability is a hard quality to rigorously define for an agent in a computer game, and it varies from game to game (particularly as the modalities of interaction with other game agents change). In Section 4.2.2, it was claimed that by using an anytime planner an agent could be given more human-like, believable weaknesses. An interesting direction for future work would be to attempt to determine the validity of this in a variety of game worlds. One possible way to determine how believable an agent is was offered by the group developing the Soar Quakebot (see Section 7.3.1). They got many players to play against different instances of their agent and then questioned the players afterwards. Such an investigation would be possible using the example agent developed as a proof-of-concept implementation of the agent architecture design. In particular, players could be asked to play against the different agents produced by the various iterations of the design process in Chapter 8 and offer opinions on how the believability of the agent changes as alterations are made to the design of the architecture.

Appendices

Appendix A

The Capture The Flag Planning Domain

The primitive actions from the Capture The Flag domain are as follows. They are expressed as a series of lists in the form;

```
[[<name> <parameters>]
   [<preconditions>]
   [<effects>]
]
```

The add and delete lists from standard STRIPS operators are combined in one effects list, with the add effects being normal atoms and the delete effects being negated atoms. Preconditions, and add and delete lists were introduced in Section 3.1.

```
[[DO_NOTHING]
   []
   []
]

[[pickup ?_agent ?_object ?_pathNode]
   [[this ?_agent]
    [at ?_agent ?_pathNode]
    [at ?_object ?_pathNode]]
   [[have ?_agent ?_object]]
]
```

```
[[putdown ?_agent ?_object ?_pathNode]
  [[this ?_agent]
   [at ?_agent ?_pathNode]
   [have ?_agent ?_object]]
  [[not [have ?_agent ?_object]]
   [at ?_object ?_pathNode]]
]

[[changeweapon ?_agent ?_old_gun ?_gun]
  [[this ?_agent]
   [have ?_agent ?_gun]
   [armedwith ?_agent ?_old_gun]
   [weapon ?_gun ?_dont_matter]]
  [[armedwith ?_agent ?_gun]
   [not [armedwith ?_agent ?_old_gun]]]
]

[[planpath ?_agent ?_position ?_destination]
  [[this ?_agent]
   [at ?_agent ?_position]
   [not [knowpath ?_agent ?_position ?_destination]]]
  [[knowpath ?_agent ?_position ?_destination]
   [knowpath ?_agent ?_destination ?_position]]
]

[[followpathnodes ?_agent ?_position ?_destination]
  [[this ?_agent]
   [knowpath ?_agent ?_position ?_destination]
   [at ?_agent ?_position]]
  [[at ?_agent ?_destination]
   [not [at ?_agent ?_position]]]
]

[[getinrange ?_this ?_thing]
  []
  [[near ?_this ?_thing]]
]
```

```
[[engage ?_this ?_agent]
  [[near ?_this ?_agent]]
  []
]
```

The methods in the Capture The Flag domain are presented using the following representation;

```
[[<name>]
  [<reduction>]
  [<constraints>]
]
```

The name of the method represents the task that it is a reduction for (e.g. `[have ?_this ?_item]` is the method to reduce an abstract `have` task like `[n1 have me flag]`). The reduction of the method is the tasks that are put into the task network in place of the named task being reduced (e.g. `[n1 have me flag]` is replaced by the reduction `[[n1 at me base][n2 pickup me flag base]]`). Tasks in the reduction list are identified with labels that can then be referenced by the constraints in the method. Labels take the form `nX` where `X` is an integer (e.g. `n2`). The constraints of a method represent conditions that must be made true in order for the reduction to be valid (e.g. `[< n1 n2]` means that task `n1` must occur before task `n2` in the task network).

In the methods that follow, `have` and `at` are the only methods for goal tasks. The rest are for action tasks.

```
[[scorepoint]
  [[n1 have ?_this ?_flag][n2 scoreflag ?_flag]]
  [
    and
    [< n1 n2]
    [before [at ?_flag ?_place] n1]
    [after [at ?_this ?_place] n1]
    [initially [this ?_this]]
    [initially [opposingteam ?_opTeam]]
    [initially [teamflag ?_opTeam ?_flag ?_state]]
  ]
]
```

```
[[defendbase]
  [[n1 attack ?_threat]]
  [
    and
    [initially [this ?_this]]
    [initially [agentteam ?_agTeam]]
    [initially [teamflag ?_agTeam ?_flag ?_state]]
    [before [at ?_flag ?_pathNode] n1]
    [before [near ?_threat ?_flag] n1]
  ]
]

[[defendbase]
  [[n1 at ?_this ?_pathNode]]
  [
    and
    [initially [this ?_this]]
    [initially [agentteam ?_agTeam]]
    [initially [teambase ?_agTeam ?_pathNode]]
  ]
]

[[defendflagcarrier]
  [[n1 attack ?_threat]]
  [
    and
    [initially [opposingteam ?_opTeam]]
    [initially [teamflag ?_opTeam ?_flag ?_state]]
    [initially [agentteam ?_myTeam]]
    [initially [onteam ?_teammate ?_myTeam]]
    [initially [onteam ?_threat ?_opTeam]]
    [before [have ?_teammate ?_flag] n1]
    [before [near ?_threat ?_teammate] n1]
  ]
]
```

```
[[defendflagcarrier]
  [[n1 getinrange ?_this ?_teammate]]
  [
    and
    [initially [this ?_this]]
    [initially [opposingteam ?_opTeam]]
    [initially [teamflag ?_opTeam ?_flag ?_state]]
    [initially [agentteam ?_myTeam]]
    [initially [onteam ?_teammate ?_myTeam]]
    [before [have ?_teammate ?_flag] n1]
    [not [before [near ?_threat ?_teammate] n1]]
  ]
]
[[interceptflagcarrier]
  [[n1 attack ?_threat]]
  [and
    and
    [initially [agentteam ?_agTeam]]
    [initially [teamflag ?_agTeam ?_flag ?_state]]
    [initially [onteam ?_threat ?_otherTeam ]]
    [not [veq ?_agTeam ?_otherTeam]]
    [before [have ?_threat ?_flag] n1]
  ]
]

[[interceptflagcarrier]
  [[n1 at ?_this ?_base]]
  [
    and
    [initially [this ?_this]]
    [initially [opposingteam ?_opTeam]]
    [initially [teambase ?_opTeam ?_base]]
  ]
]
```

```
[[heal]
  [[n1 have ?_this ?_health_id]]
  [
    and
    [initially [this ?_this]]
    [before [health ?_health_id Botpack_MedBox] n1]
    [before [near ?_this ?_health_id] n1]
  ]
]

[[heal]
  [[n1 have ?_this ?_armour_id]]
  [
    and
    [initially [this ?_this]]
    [before [armour ?_armour_id full] n1]
    [before [near ?_this ?_armour_id] n1]
  ]
]

[[heal]
  [[n1 have ?_this ?_health_id]]
  [
    and
    [initially [this ?_this]]
    [before [health ?_health_id Botpack_MedBox] n1]
    [initially [nearest_health ?_health_id]]
  ]
]

[[returnflag]
  [[n1 have ?_agent ?_flag]]
  [
    and
    [initially [this ?_agent]]
    [initially [agentteam ?_team]]
    [initially [teamflag ?_team ?_flag ?_state]]
  ]
]
```

```
[[at ?_this ?_dest]
  [[n1 DO_NOTHING]]
  [
    and
    [initially [this ?_this]]
    [before [at ?_this ?_dest] n1]
  ]
]

[[at ?_this ?_dest]
  [
    [n1 knowpath ?_this ?_pos ?_dest]
    [n2 followpathnodes ?_this ?_pos ?_dest]
  ]
  [
    and
    [initially [this ?_this]]
    [not [veq ?_pos ?_dest]]
    [not [before [at ?_this ?_dest] n1]]
    [before [at ?_this ?_pos] n2]
    [< n1 n2]
  ]
]

[[knowpath ?_this ?_pos ?_dest]
  [[n1 DO_NOTHING]]
  [[before [knowpath ?_this ?_pos ?_dest] n1]]
]

[[knowpath ?_this ?_pos ?_dest]
  [[n1 planpath ?_this ?_pos ?_dest]]
  [[not [before [knowpath ?_this ?_pos ?_dest] n1]]]
]
```

```
[[have ?_this ?_item]
  [[n1 DO_NOTHING]]
  [
    and
    [initially [this ?_this]]
    [before [have ?_this ?_item] n1]
  ]
]

[[have ?_this ?_item]
  [
    [n1 at ?_this ?_pos]
    [n2 pickup ?_this ?_item ?_pos]
  ]
  [
    and
    [not [before [have ?_this ?_item] n1]]
    [initially [this ?_this]]
    [before [at ?_item ?_pos] n1]
    [between [at ?_this ?_pos] n1 n2]
    [< n1 n2]
  ]
]

[[scoreflag ?_flag]
  [
    [n1 at ?_this ?_base]
    [n2 putdown ?_flag ?_base]
  ]
  [
    and
    [< n1 n2]
    [initially [this ?_this]]
    [initially [agentteam ?_agTeam]]
    [initially [teambase ?_agTeam ?_base]]
  ]
]
```

```
[[attack ?_agent]
  [
    [n1 getinrange ?_this ?_agent]
    [n2 armedwith ?_thisrifle]
    [n3 engage ?_this ?_agent]
  ]
  [
    and
    [< n1 n3]
    [< n2 n3]
    [initially [this ?_this]]
    [before [armedwith ?_this ?_weapon] n3]
    [not [veq ?_weapon Nothing]]
    [before [near ?_this ?_agent] n3]
  ]
]

[[armedwith ?_this ?_weapon]
  [[n1 DO_NOTHING]]
  [[before [armedwith ?_this ?_weapon] n1]]
]

[[armedwith ?_this ?_weapon]
  [[n1 changeweapon ?_this ?_old ?_new]]
  [
    and
    [initially [this ?_this]]
    [not [veq ?_old ?_new]]
    [before [armedwith ?_this ?_old] n1]
    [before [have ?_this ?_new] n1]
    [before [weapon ?_new ?_type ] n1]
  ]
]
```

Appendix B

The Blocks World Planning Domain UMCP Notation

The primitive actions from the Blocks World domain are as follows. They are expressed as a series of lists in the form;

```
[[<name> <parameters>]
   [<preconditions>]
   [<effects>]
]
```

The add and delete lists from standard STRIPS operators are combined in one effects list, with the add effects being normal atoms and the delete effects being negated atoms. These terms were introduced in Section 3.1.

```
[[DO_NOTHING]
   []
   []
]

[[unstack ?_x ?_y]
   [[clear ?_x]
    [on ?_x ?_y]]
   [[not[on ?_x ?_y]]
    [onTable ?_x]
    [clear ?_y]]
]
```

```
[[dostack ?_x ?_y]
  [[clear ?_x]
   [onTable ?_x]
   [clear ?_x]]
  [[not [onTable ?_x]]
   [on ?_x ?_y]
   [not [clear ?_y]]]
]

[[restack ?_x ?_y ?_z]
  [[clear ?_x]
   [on ?_x ?_y]
   [clear ?_z]]
  [[not [on ?_x ?_y]]
   [not[clear ?_z]]
   [clear ?_y]
   [on ?_x ?_z]]
]
```

The methods in the Capture The Flag domain are presented using the following representation;

```
[[<name>]
  [<reduction>]
  [<constraints>]
]
```

The different parts of a method were described in Appendix A. In the methods that follow, `on` and `clear` are methods for reducing goal tasks.

```
[[clear ?_x]
  [[n1 DO_NOTHING]]
  [[before [clear ?_x] n1]]
]
```

```
[[clear ?_x]
  [[n1 clear ?_y] [n2 unstack ?_y ?_x]]
  [
    and
    [< n1 n2]
    [between [clear ?_y] n1 n2]
    [before [on ?_y ?_x] n2]
    [after [clear ?_x] n2]]
  ]
]

[[on ?_x ?_y]
  [[n1 DO_NOTHING]]
  [[before [on ?_x ?_y] n1]]
]

[[on ?_x ?_y]
  [
    [n1 clear ?_x]
    [n2 clear ?_y]
    [n3 restack ?_x ?_z ?_y]
  ]
  [
    and
    [< n1 n3]
    [< n2 n3]
    [not [veq ?_x ?_y]]
    [not [veq ?_y ?_z]]
    [not [veq ?_x ?_z]]
    [before [not [on ?_x ?_y]] n3]
    [before [not [onTable ?_x]] n3]
    [before [on ?_x ?_z] n3 ]
    [between [clear ?_x] n1 n3]
    [between [clear ?_y] n2 n3]
    [between [on ?_x ?_z] n1 n3]
    [after [on ?_x ?_y] n3]]
  ]
]
```

```
[[on ?_x ?_y]
  [
    [n1 clear ?_x]
    [n2 clear ?_y]
    [n3 dostack ?_x ?_y]
  ]
  [
    and
    [< n1 n3]
    [< n2 n3]
    [not [veq ?_x ?_y]]
    [before [onTable ?_x] n3]
    [between [clear ?_x] n1 n3]
    [between [clear ?_y] n2 n3]
    [after [on ?_x ?_y] n3]]
  ]
]
```

Appendix C

An Example of the HSP Heuristic

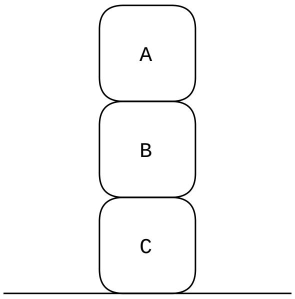

Figure C.1: An example state in the Blocks World.

This Appendix presents an example of the HSP heuristic being used to calculate the heuristic costs for a particular state in the Blocks World. The operators and representation used are presented in Appendix B. The state on which the heuristic is used is depicted in Figure C.1 and is represented by Figure C.2.

```
[[on A B][on B C][clear A][onTable C]]
```

Figure C.2: Current state after no iterations.

We will be applying the heuristic to calculate the costs of any possible `on` and `clear` atoms. We focus on these because they are the goals of the two goal tasks from the Blocks World planning domain. Given the above state, the atoms we can calculate values for are shown in Table C.1. For atoms that are in the initial state we set their costs to zero (as they have taken no iterations to add to the state and are therefore automatically achieved). As we don't know the costs of the other atoms yet, we set their cost values to infinity. This reflects the heuristic algorithm from Section 5.3.

From this starting point we attempt to apply all of the available planning operators to the state. This adds more atoms to the state. After all the operators have been applied, and new atoms added to the state have their costs set to the number of cycles it took to introduce them (see Section 5.2 for a full explanation of this). Atoms are never removed from the state because the delete lists of operators are ignored. For each iteration the applicable operators are presented, followed by the resulting state and the

Atom	**Cost**
`[on A B]`	0
`[on A C]`	∞
`[on B A]`	∞
`[on B C]`	0
`[on C A]`	∞
`[on C B]`	∞
`[clear A]`	0
`[clear B]`	∞
`[clear C]`	∞

Table C.1: Atom costs after no iterations.

new table of costs. Only the operators that add new atoms to the state are shown. New additions to the state are placed at the top of the list, and alterations to the cost table are be presented in bold.

Operator	**New Atom**
`[unstack A B]`	`[clear B]`

Table C.2: Applicable operators in iteration one.

```
[[clear B][onTable A]
[on A B][on B C][clear A][onTable C]]
```

Figure C.3: Current state after one iteration.

Atom	**Cost**
`[on A B]`	0
`[on A C]`	∞
`[on B A]`	∞
`[on B C]`	0
`[on C A]`	∞
`[on C B]`	∞
`[clear A]`	0
`[clear B]`	**1**
`[clear C]`	∞

Table C.3: Atom costs after one iteration.

Operator	**New Atom**
[unstack B C]	[clear C]
[restack B C A]	[on B A]

Table C.4: Applicable operators in iteration two.

```
[[clear C][on B A][onTable B]
[clear B][onTable A][on A B][on B C]
[clear A][onTable C]]
```

Figure C.4: Current state after two iterations.

Atom	**Cost**
[on A B]	0
[on A C]	∞
[on B A]	**2**
[on B C]	0
[on C A]	∞
[on C B]	∞
[clear A]	0
[clear B]	1
[clear C]	**2**

Table C.5: Atom costs after two iterations.

Operator	**New Atom**
[dostack A C]	[on A C]
[dostack C A]	[on C A]
[dostack C B]	[on C B]

Table C.6: Applicable operators in iteration three.

```
[on A C][on C A][on C B]
[clear C][on B A][onTable B][clear B]
[onTable A][on A B][on B C][clear A]
[onTable C]]
```

Figure C.5: Current state after three iterations.

Atom	**Cost**
`[on A B]`	0
`[on A C]`	**3**
`[on B A]`	2
`[on B C]`	0
`[on C A]`	**3**
`[on C B]`	**3**
`[clear A]`	0
`[clear B]`	1
`[clear C]`	2

Table C.7: Atom costs after three iterations.

Once the cost table is filled (see Table C.7) the iterations end. The end point is defined as when the cost lists in adjacent iterations are identical. Because costs cannot be reduced by further iterations (they are always the minimum possible value), when all values are calculated the iterations end. This would be different if some atoms were unreachable from the starting state.

Appendix D

An Example A-UMCP Planning Process

This Appendix presents a slightly abridged example of a typical A-UMCP planning process. The example uses the Blocks World planning domain presented in Appendix B. The example demonstrates how A-UMCP reduces an abstract description of a problem into a primitive plan. The example also demonstrates the way in which the planning process produces plans with improved quality as measured by the heuristic described in Section 5.2. The planning example is abridged to focus mainly on the reduction parts of the planning process. The majority of the constraint resolution steps are ignored to allow for a clearer overview of the planning process.

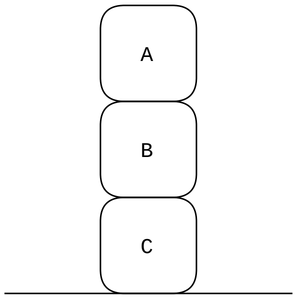

Figure D.1: The example initial state.

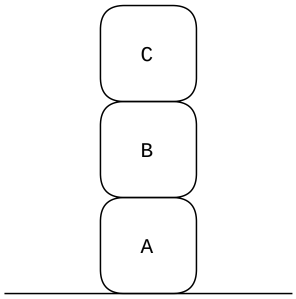

Figure D.2: The goal state.

The initial state for the planning process is shown in Figure D.1. The goal state the planning process is trying to reach is shown in Figure D.2. This goal state can be represented by the following task network.

`[[on B A][on C B]]`

The two `on` tasks are abstract representations of the actions needed to achieve the goals they represent. This example uses a simplified representation for task networks. The tasks in the network are shown as an ordered list (as in the previous example). The order of the tasks in the list represents the ordering constraints on the tasks which, along with other constraints, are left out from this example for conciseness. Full details of which constraints are applicable to which reductions can be seen in the full planning domain in Appendix B. How these constraints are resolved is determined by the algorithms presented in [Erol, 1995, Chapter 6]. The following example is composed of steps that reduce a task from a task network to create successor task networks, followed by the selection of a particular task network to reduce next.

We only have one task network to start with (the one representing the goal state). For this task network we must first calculate its heuristic cost. The initial state is the same as the state used in Appendix C to demonstrate the behaviour of the heuristic, so for this particular task network we can use the results stated in Table C.7. In our starting task network `[on B A]` has a cost of 2 and `[on C B]` has a cost of 3, so our starting task network has a total cost of 5.

We can now select a task for reduction from our task network. At this point we select the first task (`[on B A]`) for reduction. The methods for reduction give us three possible reductions. These are;

	Task Network	**Cost**
1	`[DO_NOTHING]` `[on C B]`	3
2	`[clear B][clear A][restack B C A]` `[on C B]`	4
3	`[clear B][clear A][dostack B A]` `[on C B]`	4

The constants are assigned to the task variables during the (ignored) constraint refinement step. Costs are calculated by the heuristic method presented in Chapter 5.2 and demonstrated in Appendix C. In this instance the costs are the same as those generated by the initial state. This is because even though the new primitives could change the state, they are not actually applicable because their preconditions remain unsatisfied. From these reductions, the task with the lowest cost should be selected for refinement. In this case, because the reduction into a `DO_NOTHING` task is not valid (i.e. `[on B A]` is not already true), it is removed from the search space by the `DO_NOTHING` critic (see Section 6.5). We are then left with two reductions of the same cost which have been introduced into the search space at the same time. In the actual planner, the order in which they would be considered depends on the order in which the reduction methods were applied. In this example, because we know the `restack` reduction leads to a solution faster, we shall choose to reduce this task network and ignore the other. Again we choose the first abstract task to reduce. This gives us;

	Task Network	**Cost**
1	[DO_NOTHING][clear A] [restack B C A][on C B]	3
2	[clear A][unstack A B][clear A] [restack B C A][on C B]	1

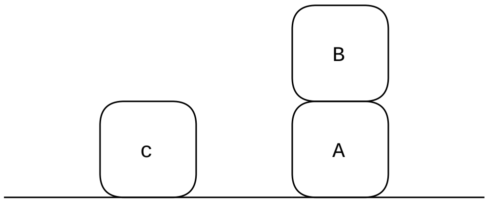

Figure D.3: The intermediate state.

From these reductions the invalid DO_NOTHING reduction is again removed, leaving us with only the second reduction. The cost of the second reduction is generated using a new cost list (i.e. not the one from the initial state) because the introduction of unstack primitive establishes the preconditions of the restack action, and both primitives change the state. This leaves us with the state depicted by Figure D.3. Reducing the first task of the remaining task network gives us;

	Task Network	**Cost**
1	[DO_NOTHING] [unstack A B][clear A][restack B C A] [on C B]	1
2	[clear ??][unstack ?? A] [unstack A B][clear A][restack B C A] [on C B]	1

Because nothing is on top of block A at the start of the plan, the second reduction is invalid and is removed by the constraint refinement procedure. We can now reduce the first abstract ([clear A]) task from the first reduction.

	Task Network	**Cost**
1	[DO_NOTHING][unstack A B] [DO_NOTHING][restack B C A] [on C B]	1
2	[DO_NOTHING][unstack A B] [clear ??][unstack ?? A] [restack B C A][on C B]	1

This situation is almost identical to the previous one, and the second reduction can be ignored. The first reduction now gives us a complete first half of a plan. Block B has been placed on block A and there are no abstract tasks left in that part of the plan. We can now go on to reduce the remaining abstract task, [on C B]. This produces the reductions;

	Task Network	**Cost**
1	[DO_NOTHING][unstack A B] [DO_NOTHING][restack B C A][DO_NOTHING] [DO_NOTHING]	0
2	[DO_NOTHING][unstack A B] [DO_NOTHING][restack B C A] [clear C][clear B][restack C ?? B]	0
3	[DO_NOTHING][unstack A B] [DO_NOTHING][restack B C A] [clear C][clear B][dostack C B]	0

The first and second reductions can be ignored for previously cited reasons. The third reduction is now an almost valid plan for inverting the tower of blocks. The remaining abstract tasks ([clear C] and [clear B]) have valid DO_NOTHING reductions and will be simply reduced into these.

Appendix E

Problem Definitions

This appendix contains the example problems used throughout Chapter 6.

E.1 Blocks World Problems

Problem for Table 6.1.

Initial State:

```
[[clear block1][clear block3]
 [on block1 block2][on block3 block4][on block4 block5]
 [onTable block2][onTable block5]]
```

Goal State:

```
[[n1 on block4 block1][n2 on block2 block3]
 [n3 on block3 block5]]
[[< n3 n2][< n2 n1]]
```

Problem for Figure 6.1.

Initial State:

```
[[clear block1][clear block3][clear block4]
 [on block3 block2]
 [onTable block4][onTable block2][onTable block1]]
```

Goal State:

```
[[n1 on block2 block1]
 [n2 on block3 block2]
 [n3 on block4 block3]]
[[< n3 n2][< n2 n1]]
```

Problem for Figure 6.4.

Initial State:

```
[[clear block1][clear block2][clear block3]]
 [clear block4][clear block5][clear block6]
 [onTable block6][onTable block5][onTable block4]
 [onTable block3][onTable block2][onTable block1]]
```

Goal State:

```
[[n1 on block2 block1]
 [n2 on block3 block2]
 [n3 on block4 block3]
 [n4 on block5 block4]
 [n5 on block6 block5]]
[[< n1 n2][< n2 n3][< n3 n4][< n4 n5]]
```

Problem for Figure 6.5.

Initial State:

```
[[clear block1]]
 [on block1 block2]
 [onTable block2]]
```

Goal State:

```
[[n1 on block2 block1]]
[]
```

Problem for Figure 6.6.

Initial State:

```
[[clear block4]
 [on block4 block3][on block3 block2]
 [on block2 block1]
 [onTable block1]]
```

Goal State:

```
[[n1 on block3 block4]
 [n2 on block2 block3]
 [n3 on block1 block2]]
[[< n1 n2][< n2 n3]]
```

Problem for Figure 6.7.

Initial State:

```
[[clear block1]
 [on block1 block2][on block2 block3]
 [onTable block3]]
```

Goal State:

```
[[n1 on block2 block1]
 [n2 on block3 block2]]
[[< n1 n2]]
```

Problem for Figure 6.12.

Initial State:

```
[[clear block6]
 [on block2 block1][on block3 block2]
 [on block4 block3][on block5 block4]
 [on block6 block5]
 [onTable block1]]
```

Goal State:

```
[[n1 clear block1]]
[]
```

Problem for Figure 6.13.

Initial State:

```
[[clear block6]
 [on block2 block1][on block3 block2]
 [on block4 block3][on block5 block4]
 [on block6 block5][on block7 block6]
 [on block8 block7]
 [onTable block1]]
```

Goal State:

```
[[n1 clear block1]]
[]
```

E.2 Capture The Flag Problems

Problem for Table 6.2.

Initial State:

```
[[agentteam Blue][opposingteam Red]
 [onteam FemBot1 Blue][onteam FemBot2 Blue]
 [onteam MaleBot1 Red][onteam MaleBot2 Red]
 [teamflag Red RedFlag Home][teambase Red RedFlagBase]
 [teamflag Blue BlueFlag Home][teambase Blue BlueFlagBase]
 [weapon Enforcer1 Enforcer][weapon FlakCannon1 FlakCannon]
 [health MedBox1 MedBox]
 [at FemBot1 PlayerStart3][at FemBot2 PlayerStart4]
 [at MaleBot1 PlayerStart1][at MaleBot2 PlayerStart2]
 [at RedFlag RedFlagBase][at BlueFlag BlueFlagBase]
 [at MedBox1 InventorySpot2][at FlakCannon1 InventorySpot1]
 [this FemBot1][armedwith FemBot1 Enforcer1]
 [have FemBot1 Enforcer1][nearest_health MedBox1]]
```

Goal State:

```
[[n1 pickup FemBot1 RedFlag RedFlagBase]
 [n2 at FemBot1 RedFlagBase]
 [n3 at FemBot1 BlueFlagBase]]
[[< n2 n3] [< n1 n3]]
```

Problem for Figure 6.2.

Initial State:

```
[[agentteam Blue][opposingteam Red]
 [onteam FemBot1 Blue]
 [teamflag Red RedFlag Home] [teambase Red RedFlagBase]
 [teamflag Blue BlueFlag Home][teambase Blue BlueFlagBase]
 [at FemBot1 PlayerStart4][at RedFlag RedFlagBase]
 [at BlueFlag BlueFlagBase][this FemBot1]]
```

Goal State:

```
[[n1 scorepoint]]
[]
```

Problem for Figure 6.10.

Initial State:

```
[[agentteam Blue][opposingteam Red]
 [onteam FemBot1 Blue]
 [teamflag Red RedFlag Home]
 [teambase Red RedFlagBase]
 [teamflag Blue BlueFlag Home]
 [teambase Blue BlueFlagBase]
 [at FemBot1 PlayerStart4][at RedFlag RedFlagBase]
 [at BlueFlag BlueFlagBase]
 [this FemBot1]]
```

Goal State:

```
[[n1 scorepoint]]
[]
```

Problem for Figure 6.11.

Initial State:

```
[[agentteam Blue][opposingteam Red]
 [onteam FemBot1 Blue][onteam FemBot2 Blue]
 [onteam MaleBot3 Blue][onteam MaleBot1 Red]
 [onteam MaleBot2 Red][onteam FemBot3 Red]
 [teamflag Red RedFlag Home]
 [teambase Red RedFlagBase]
 [teamflag Blue BlueFlag Home]
 [teambase Blue BlueFlagBase]
 [weapon Enforcer1 Enforcer]
 [weapon FlakCannon1 FlakCannon]
 [weapon FlakCannon2 FlakCannon]
 [weapon FlakCannon3 FlakCannon]
 [health MedBox1 MedBox][health MedBox2 MedBox]
 [at FemBot1 PlayerStart4][at FemBot2 PlayerStart5]
 [at MaleBot3 PlayerStart6][at MaleBot1 PlayerStart1]
 [at MaleBot2 PlayerStart2][at FemBot3 PlayerStart3]
 [at RedFlag RedFlagBase][at BlueFlag BlueFlagBase]
 [at MedBox1 InventorySpot2][at MedBox2 InventorySpot4]
 [at FlakCannon1 InventorySpot1]
 [at FlakCannon2 InventorySpot3]
 [at FlakCannon3 InventorySpot5]
 [this FemBot1]
 [armedwith FemBot1 Enforcer1]
```

```
 [have FemBot1 Enforcer1]
 [nearest_health MedBox2]]
```

Goal State:

```
[[n1 scorepoint]]
[]
```

E.3 UM Translog Problems

In all of the UM Translog problems, the following facts are always in the initial state (in addition to those added for each specific problem).

```
[[rv_compatible air_route airplane]
 [rv_compatible rail_route traincar]
 [rv_compatible rail_route train]
 [rv_compatible road_route truck]
 [pv_compatible cars auto]
 [pv_compatible livestock livestock]
 [pv_compatible mail airplane]
 [pv_compatible mail mail]
 [pv_compatible valuable armored]
 [pv_compatible perishable refrigerated]
 [pv_compatible granular hopper]
 [pv_compatible liquid tanker]
 [pv_compatible bulky flatbed]
 [pv_compatible regular airplane]
 [pv_compatible regular mail]
 [pv_compatible regular flatbed]
 [pv_compatible regular regular]
 [type region1 region][type region2 region]
 [pc_compatible city1 hazardous]
 [in_region city1 region1][type city1 city]
 [not [type city1_cl1 tcenter]]
 [in_city city1_cl1 city1][type city1_cl1 clocation]
 [not [type city1_cl2 tcenter]]
 [in_city city1_cl2 city1][type city1_cl2 clocation]
 [serves city1_ts1 city1][not [type city1_ts1 hub]]
 [available city1_ts1][type city1_ts1 tcenter]
 [in_city city1_ts1 city1][type city1_ts1 train_station]
```

```
[serves city1_ts2 city1] [not [type city1_ts2 hub]]
[available city1_ts2][type city1_ts2 tcenter]
[in_city city1_ts2 city1][type city1_ts2 train_station]
[serves city1_ap1 city1]not [type city1_ap1 hub]]
[available city1_ap1][type city1_ap1 tcenter]
[in_city city1_ap1 city1][type city1_ap1 airport]
[serves city1_ap2 city1][not [type city1_ap2 hub]]
[available city1_ap2][type city1_ap2 tcenter]
[in_city city1_ap2 city1][type city1_ap2 airport]
[pc_compatible city2 hazardous][in_region city2 region2]
[type city2 city][not [type city2_cl1 tcenter]]
[in_city city2_cl1 city2][type city2_cl1 clocation]
[serves city2_ap1 city2][not [type city2_ap1 hub]]
[available city2_ap1][type city2_ap1 tcenter]
[in_city city2_ap1 city2][type city2_ap1 airport]
[serves city2_ts1 city2][not [type city2_ts1 hub]]
[available city2_ts1][type city2_ts1 tcenter]
[in_city city2_ts1 city2][type city2_ts1 train_station]
[pc_compatible city3 hazardous]
[in_region city3 region1][type city3 city]
[not [type city3_cl1 tcenter]][in_city city3_cl1 city3]
[type city3_cl1 clocation] [serves city3_ap1 city3]
[not [type city3_ap1 hub]] [available city3_ap1]
[type city3_ap1 tcenter] [in_city city3_ap1 city3]
[type city3_ap1 airport][serves city3_ts1 city3]
[not [type city3_ts1 hub]] [available city3_ts1]
[type city3_ts1 tcenter] [in_city city3_ts1 city3]
[type city3_ts1 train_station][serves region1_ap1 region1]
[type region1_ap1 hub] [available region1_ap1]
[type region1_ap1 tcenter][in_city region1_ap1 city2]
[type region1_ap1 airport][serves region1_ts1 region1]
[type region1_ts1 hub][available region1_ts1]
[type region1_ts1 tcenter][in_city region1_ts1 city3]
[type region1_ts1 train_station]
[available road_route_1]
[connects road_route_1 road_route city3 city1]
[connects road_route_1 road_route city1 city3]
[available road_route_2]
[connects road_route_2 road_route city3 city2]
```

```
[connects road_route_2 road_route city2 city3]
[available air_route_1]
[connects air_route_1 air_route city2_ap1 city1_ap1]
[connects air_route_1 air_route city1_ap1 city2_ap1]
[available air_route_2]
[connects air_route_2 air_route region1_ap1 city1_ap1]
[connects air_route_2 air_route city1_ap1 region1_ap1]
[available air_route_3]
[connects air_route_3 air_route region1_ap1 city3_ap1]
[connects air_route_3 air_route city3_ap1 region1_ap1]
[available air_route_4]
[connects air_route_4 air_route city1_ap2 city1_ap1]
[connects air_route_4 air_route city1_ap1 city1_ap2]
[available rail_route_1]
[connects rail_route_1 rail_route city1_ts2 city1_ts1]
[connects rail_route_1 rail_route city1_ts1 city1_ts2]
[available rail_route_2]
[connects rail_route_2 rail_route city2_ts1 city1_ts1]
[connects rail_route_2 rail_route city1_ts1 city2_ts1]
[available rail_route_3]
[connects rail_route_3 rail_route region1_ts1 city1_ts1]
[connects rail_route_3 rail_route city1_ts1 region1_ts1]
[available rail_route_4]
[connects rail_route_4 rail_route region1_ts1 city3_ts1]
[connects rail_route_4 rail_route city3_ts1 region1_ts1]
[ramp_available ramp1a][available ramp1a]
[at_equipment ramp1a city1_ap1][type ramp1a plane_ramp]
[ramp_available ramp1b][available ramp1b]
[at_equipment ramp1b city1_ap2][type ramp1b plane_ramp]
[ramp_available ramp2][available ramp2]
[at_equipment ramp2 city3_ap1] [type ramp2 plane_ramp]
[ramp_available ramp3][available ramp3]
[at_equipment ramp3 city2_ap1][type ramp3 plane_ramp]
[ramp_available ramp4][available ramp4]
[at_equipment ramp4 region1_ap1] [type ramp4 plane_ramp]
[available road_route_i1547]
[connects road_route_i1547 road_route city2 city1]
[available road_route_i1548]
[connects road_route_i1548 road_route city1 city2]]
```

Problem for Figures 6.3 and 6.9.
Initial State:
```
[[at_package pkg_1 city1_cl1][type pkg_1 regular]
 [not [type pkg_1 hazardous]]
 [not [type pkg_1 valuable]]
 [guard_inside truck_1][available truck_1]
 [not [type truck_1 traincar]]
 [at_vehicle truck_1 city1_cl2]
 [type truck_1 armored][type truck_1 regular]
 [type truck_1 truck]]
```

Goal State:
```
[[n1 carry pkg_1 city1_cl1 city1_cl2]]
[]
```

Problem for Figure 6.8.
Initial State:
```
[[at_package pkg_1 city1_cl1][type pkg_1 regular]
 [not [type pkg_1 hazardous]]
 [not [type pkg_1 valuable]]
 [guard_inside truck_1][available truck_1]
 [not [type truck_1 traincar]]
 [at_vehicle truck_1 city1_cl1]
 [type truck_1 armored][type truck_1 regular]
 [type truck_1 truck]]
```

Goal State:
```
[[n1 carry pkg_1 city1_cl1 city1_cl2]]
[]
```

Bibliography

[Aamodt and Plaza, 1994] Aamodt, A. and Plaza, E. (1994). Case-based reasoning: Foundational issues, methodological variations, and system approaches. *Artificial Intelligence Communications*, 7(1):39 – 59.

[Agre and Chapman, 1987] Agre, P. E. and Chapman, D. (1987). Pengi: An implementation of a theory of activity. In *Proceedings of The Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 268–272, Seattle, WA, USA. AAAI.

[Andrews et al., 1995] Andrews, S., Kettler, B., Erol, K., and Hendler, J. (1995). UM Translog: A planning domain for the development and benchmarking of planning systems. Technical Report CS-TR-3487, University of Maryland, Department of Computer Science.

[Atkin and Cohen, 1998] Atkin, M. S. and Cohen, P. R. (1998). Physical planning and dynamics. In *Working Notes of the AAAI Fall Symposium on Distributed Continual Planning*, pages 4 –9. AAAI.

[Atkin et al., 1999] Atkin, M. S., Westbrook, D. L., and Cohen, P. R. (1999). Capture the flag: Military simulation meets computer games. In *Papers from the 1999 AAAI Spring Symposium on Artificial Intelligence and Computer Games*, pages 1–5. AAAI.

[Aylett et al., 1997] Aylett, R. S., Coddington, A. M., Barnes, D. P., and Ghanea-Hercock, R. (1997). What does a planner need to know about execution? In Steel, S. and Alami, R., editors, *Recent advances in AI planning*, volume 1348 of *Lecture Notes in Computer Science*, pages 26–38. Springer.

[Aylett et al., 1995] Aylett, R. S., Coddington, A. M., Barnes, D. P., Ghanea-Hercock, R., and Gray, J. O. (1995). Planning and behaviours - a hybrid architecture for mobile robots. In *Proceedings, 2nd IFAC Conference on Intelligent Autonomous Vehicles*, Espoo, Finland.

[Aylett et al., 1999] Aylett, R. S., Horrobin., A. J., O'Hare, J. J., Osman, A. A., and Polshaw, M. M. T. (1999). Virtual teletubbies: reapplying a robot architecture to virtual agents. In Etzioni, O., Müller, J. P., and Bradshaw, J. M., editors, *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pages 338–339, Seattle, WA, USA. ACM Press.

[Beaudoin, 1994] Beaudoin, L. P. (1994). *Goal Processing In Autonomous Agents*. PhD thesis, School of Computer Science, The University of Birmingham.

[BioWare, 1998] BioWare (1998). *Baldur's Gate*. Interplay.

[Blizzard, 2002] Blizzard (2002). *Warcraft III: Reign of Chaos*. Blizzard.

[Blum and Furst, 1997] Blum, A. and Furst, M. (1997). Fast planning through graph analysis. *Artificial Intelligence*, 90:281 – 300.

[Blythe, 1999] Blythe, J. (1999). An overview of planning under uncertainty. *AI Magazine*, 20(2):37–54.

[Blythe and Reilly, 1993] Blythe, J. and Reilly, W. S. (1993). Integrating reactive and deliberative planning for agents. Technical report, School of Computer Science, Carnegie Mellon University. CMU-CS-93-155.

[Boddy and Dean, 1989] Boddy, M. and Dean, T. (1989). Solving time-dependant planning problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 979–984, Detroit, Michigan, USA. Morgan Kaufmann.

[Bonet and Geffner, 2001] Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence: Special Issue on Heuristic Search*, 129:5–33.

[Bonet et al., 1997] Bonet, B., Loerincs, G., and Geffner, H. (1997). A robust and fast action selection mechanism for planning. In *Proceedings of The Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 714–719, Providence, Rhode Island, USA. AAAI Press / The MIT Press.

[Briggs and Cook, 1999] Briggs, W. and Cook, D. (1999). Anytime planning for optimal tradeoff between deliberative and reactive planning. In *Proceedings of the Twelfth International Florida Artificial Intelligence Research Society Conference (FLAIRS-99)*, pages 367 – 370, Orlando, Florida, USA. AAAI Press.

[Brooks, 1986] Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2:14–23.

[Brooks, 1991] Brooks, R. A. (1991). Intelligence without reason. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 569–595, Sydney, New South Wales, Australia. Morgan Kaufmann.

[Bryson, 2001] Bryson, J. (2001). *Intelligence by Design*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

[Bundy, 1988] Bundy, A. (1988). The use of explicit plans to guide proofs. In *9th International Conference on Automated Deduction (CADE-9)*, volume 310 of *Lecture Notes in Computer Science*, pages 111–120, Argonne, Illinois, USA. Springer.

[Carlson and Hodgins, 1997] Carlson, D. A. and Hodgins, J. K. (1997). Simulation levels of detail for real-time animation. In Davis, W. A., Mantei, M., and Klassen, R. V., editors, *Graphics Interface '97*, pages 1–8. Canadian Human-Computer Communications Society.

[Cavazza, 2000] Cavazza, M. (2000). Merging planning and path planning: On agent's behaviours in situated virtual worlds. In *Proceedings of the AISB'00 Symposium on AI Planning and Intelligent Agents*, pages 17–24, Birmingham, UK.

[Cavazza et al., 2000] Cavazza, M., Aylett, R. S., and Dautenhahn, K. (2000). Interactive story-telling in virtual environment: Building the "holodeck". In *6th International Conference on Virtual Systems and Multi-media (VSMM 2000)*, pages 678–687, Ogaki City, Gifu Prefecture, Japan.

[Chapman, 1987] Chapman, D. (1987). Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377. Reprinted in Allen J., Hendler J. and Tate A., editors. *Readings in Planning*, pages 537-558, 1990. Morgan Kaufmann.

[Core, 2002] Core (2002). *Herdy Gerdy*. Eidos Interactive.

[Currie and Tate, 1991] Currie, K. and Tate, A. (1991). O-plan: the open planning architecture. *Artificial Intelligence*, 52:49 – 86.

[Dean and Boddy, 1988] Dean, T. and Boddy, M. (1988). An analysis of time-dependant planning. In *Proceedings of The Seventh National Conference on Artificial Intelligence (AAAI-88)*, pages 49–54, St. Paul, Minnesota, USA. AAAI Press / The MIT Press.

[Dennett, 1978] Dennett, D. C. (1978). *Brainstorms: Philosophical Essays on Mind and Psychology*. MIT Press, Cambridge, MA.

[desJardins et al., 1999] desJardins, M. E., Durfee, E. H., Ortiz, C. L., and Wolverton, M. J. (1999). A survey of research in distributed, continual planning. *AI Magazine*, 20(4):13–22.

[EA Sports, 2001] EA Sports (2001). *FIFA 2002*. EA Sports.

[Electronic Arts Pacific, 2003] Electronic Arts Pacific (2003). *Command & Conquer: Generals*. Electronic Arts.

[Ensemble Studios, 1997] Ensemble Studios (1997). *Age of Empires*. Microsoft.

[Epic Mega Games, 1999] Epic Mega Games (1999). *Unreal Tournament*. GT Interactive.

[Erol, 1995] Erol, K. (1995). *Hierarchical Task Network Planning: Formalization, Analysis, and Implementation*. PhD thesis, Department of Computer Science, The University of Maryland.

[Erol et al., 1994] Erol, K., Nau, D., and Hendler, J. (1994). UMCP: A sound and complete planning procedure for hierarchical task-network planning. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, pages 249 – 254, Chicago, Illinois, USA. AAAI Press.

[Erol et al., 1995] Erol, K., Nau, D., Hendler, J., and Tsuneto, R. (1995). A critical look at critics in HTN planning. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1592–1598, Montréal, Québec, Canada. Morgan Kaufmann.

[Ferguson, 1992] Ferguson, I. A. (1992). *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents*. PhD thesis, University of Cambridge.

[Fikes et al., 1972] Fikes, R. E., , Hart, P. E., and Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):251–288. Reprinted in Allen J., Hendler J. and Tate A., editors. *Readings in Planning*, pages 189-206, 1990. Morgan Kaufmann.

[Fikes and Nilsson, 1971] Fikes, R. E. and Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208. Reprinted in Allen J., Hendler J. and Tate A., editors. *Readings in Planning*, pages 88-97, 1990. Morgan Kaufmann.

[Firby, 1987] Firby, R. J. (1987). An investigation into reactive planning in complex domains. In *Proceedings of The Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 202–206, Seattle, WA, USA. AAAI Press.

[Firby, 1989] Firby, R. J. (1989). *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, Yale University.

[Franklin and Graesser, 1997] Franklin, S. and Graesser, A. (1997). Is it an agent, or just a program? A taxonomy for autonomous agents. In *Intelligent Agents III, Agent Theories, Architectures, and Languages*, volume 1193 of *Lecture Notes in Computer Science*, pages 21–35, Budapest, Hungary. Springer-Verlag.

[Georgeff and Ingrand, 1989] Georgeff, M. P. and Ingrand, F. F. (1989). Decision-making in an embedded reasoning system. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 972 – 978, Detroit, MI, USA. Morgan Kaufmann.

[Georgeff and Lansky, 1987] Georgeff, M. P. and Lansky, A. L. (1987). Reactive reasoning and planning. In *Proceedings of The Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle, WA, USA. AAAI Press.

[Gibson, 1986] Gibson, J. (1986). *The Ecological Approach to Visual Perception*. Lawrence Erlbaum Associates, Hillsdale, NJ. (originally published in 1979).

[Ginsberg, 1989] Ginsberg, M. L. (1989). Universal planning: An (almost) universally bad idea. *AI Magazine*, 10(4):40–44.

[Green, 1969] Green, C. (1969). Application of theorem proving to problem solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence (IJCAI-69)*, pages 219–240, Washington DC, USA. Reprinted in Allen J., Hendler J. and Tate A., editors. *Readings in Planning*, pages 67-87, 1990. Morgan Kaufmann.

[Hansen et al., 1997] Hansen, E., Zilberstein, S., and Danilchenko, V. (1997). Anytime heuristic search: First results. Technical report, Department of Computer Science, The University of Massachusetts. Technical Report UM-CS-1997-050.

[Hansen and Zilberstein, 1996] Hansen, E. A. and Zilberstein, S. (1996). Monitoring anytime algorithms. *SIGART Bulletin*, 7(2):28–33.

[Hawes, 2000] Hawes, N. (2000). Real-time goal-orientated behaviour for computer game agents. In *Game-On 2000, 1st International Conference on Intelligent Games and Simulation*, pages 71–75, London, UK.

[Hawes, 2001] Hawes, N. (2001). Anytime planning for agent behaviour. In *Proceedings of the Twentieth Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG-2001)*, pages 157–166, Edinburgh, UK.

[Hawes, 2002] Hawes, N. (2002). An anytime planning agent for computer game worlds. In *Workshop on Agents in Computer Games at The 3rd International Conference on Computers and Games (CG'02)*, pages 1–14, Edmonton, Canada.

[Hayes-Roth, 1990] Hayes-Roth, B. (1990). Architectural foundations for real-time performance in intelligent agents. *Real-Time Systems: The International Journal of Time-Critical Computing*, 2:99–125.

[Horswill and Zubeck, 1999] Horswill, I. D. and Zubeck, R. (1999). Robot architectures for believable game agents. In *Papers from the 1999 AAAI Spring Symposium on Artificial Intelligence and Computer Games*, pages 55–59.

[Howe et al., 1990] Howe, A. E., Hart, D. M., and Cohen, P. R. (1990). Addressing real-time constraints in the design of autonomous agents. Technical report, Experimental Knowledge Systems Laboratory, Department of Computer and Information Science, University of Massachusetts. COINS Technical Report 90-06.

[id Software, 1997] id Software (1997). *Quake II*. Activision.

[Isla et al., 2001] Isla, D., Burke, R., Downie, M., and Blumberg, B. (2001). A layered brain architecture for synthetic creatures. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 1051–1058, Seattle, WA, USA. Morgan Kaufmann.

[Jones et al., 1998] Jones, R. M., Laird, J. E., and Nielsen, P. E. (1998). Automated intelligent pilots for combat flight simulation. In *Proceedings of the Tenth Innovative Applications of Artificial Intelligence Conference (IAAI-98)*, pages 1047–1054, Madison, Wisconsin, USA. AAAI Press / The MIT Press.

[Kaelbling, 1990] Kaelbling, L. P. (1990). An architecture for intelligent reactive systems. In James Allen, J. H. and Tate, A., editors, *Readings in Planning*, pages 713–728. Morgan Kaufmann.

[Kaminka et al., 2002] Kaminka, G. A., Veloso, M. M., Schaffer, S., Sollitto, C., Adobbati, R., Marshall, A. N., Scholer, A., and Tejada, S. (2002). Gamebots: A flexible test bed for multiagent team research. *Communications of the ACM*, 45(1):43–45.

[Kautz and Selman, 1999] Kautz, H. and Selman, B. (1999). Unifying SAT-based and graph-based planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI 99)*, pages 318–325, Stockholm, Sweden. Morgan Kaufmann.

[Khoo et al., 2002] Khoo, A., Dunham, G., Trienens, N., and Sood, S. (2002). Efficient, realistic NPC control systems using behaviour-based techniques. In *Papers from the 2002 AAAI Spring Symposium on Artificial Intelligence and Computer Games*, pages 46–51.

[Kirsh, 1991] Kirsh, D. (1991). Today the earwig, tomorrow man? *Artificial Intelligence*, 47:161–184.

[Kitano et al., 1999] Kitano, H., Tadokoro, S., Noda, I., Matsubara, H., Takahashi, T., Shinjou, A., and Shimada, S. (1999). Robocup rescue: Search and rescue in large-scale disasters as a domain for autonomous agents research. In *Proceedings of IEEE Conference on Man, Systems, and Cybernetics, volume 6*, pages 739–743. IEEE.

[Kitano et al., 1997] Kitano, H., Tambe, M., Stone, P., Veloso, M., Coradeschi, S., Osawa, E., Matsubara, H., Noda, I., and Asada, M. (1997). The robocup synthetic agent challenge 97. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI 97)*, pages 24–30, Nagoya, Japan. Morgan Kaufmann.

[Knoblock, 1994] Knoblock, C. (1994). Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243 – 302.

[Koehler, 1998] Koehler, J. (1998). Solving complex planning tasks through extraction of subproblems. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, pages 62–69, Pittsburgh, Pennsylvania, USA. AAAI Press.

[Konami, 2001] Konami (2001). *Metal Gear Solid 2: Sons of Liberty*. Komani.

[Laird, 2000] Laird, J. E. (2000). It knows what you're going to do: Adding anticipation to a quakebot. In *Papers from the 2000 AAAI Spring Symposium on Artificial Intelligence and Computer Games*, pages 41–50.

[Laird and Duchi, 2001] Laird, J. E. and Duchi, J. C. (2001). Creating human-like synthetic characters with multiple skill levels: A case study using the soar quakebot. In *Papers from the 2001 AAAI Spring Symposium on Artificial Intelligence and Computer Games*, pages 54–58.

[Laird et al., 1987] Laird, J. E., Newell, A., and Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(3):1–64.

[Logan and Alechina, 1998] Logan, B. and Alechina, N. (1998). State space search with prioritised soft constraints. In *Proceedings of the ECAI-98 Workshop "Decision theory meets artificial intelligence: qualitative and quantitative approaches"*, pages 33–42, Brighton, UK. John Wiley and Sons.

[Logan and Poli, 1996] Logan, B. and Poli, R. (1996). Route planning in the space of complete plans. In *Proceedings of the Fifteenth Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG-1996)*, pages 233–240, Liverpool, UK.

[Long and Fox, 2003] Long, D. and Fox, M. (2003). PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*. Forthcoming in JAIR special issue on 3rd International Planning Competition. Available from http://www.dur.ac.uk/CompSci/research/stanstuff/.

[Luebke and Erikson, 1997] Luebke, D. and Erikson, C. (1997). View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH 97 Conference Proceedings*, pages 199–208, Los Angeles, California, USA. ACM.

[MacNamee et al., 2002] MacNamee, B., Dobbyn, S., Cunningham, P., and O'Sullivan, C. (2002). Men behaving appropriately: Integrating the role passing technique into the ALOHA system. In *Proceedings of the AISB'02 Symposium on Animating Expressive Characters for Social Interactions*, pages 59–62, London, UK.

[Maes, 1991] Maes, P. (1991). The agent network architecture (ANA). *SIGART Bulletin*, 2(4):115 – 120.

[Malcolm, 1997] Malcolm, C. (1997). A hybrid behavioural/knowledge-based approach to robotic assembly. In *Evolutionary Robotics: From Intelligent Robots to Artificial Life (ER'97)*, pages 221–256, Tokyo, Japan. AAI Books.

[Maxis, 2000] Maxis (2000). *The Sims*. Electronic Arts.

[McDermott, 2000] McDermott, D. (2000). The 1998 AI planning systems competition. *AI Magazine*, 2(2):35–55.

[Microsoft, 2002] Microsoft (2002). *Combat Flight Simulator 3*. Microsoft.

[Mind's Eye, 2000] Mind's Eye (2000). *Sheep*. Empire Interactive.

[Minsky, 1997] Minsky, M. (1997). A framework for representing knowledge. In Haugeland, J., editor, *Mind Design II*, pages 111–142. The MIT Press.

[Mouaddib and Zilberstein, 1995] Mouaddib, A. I. and Zilberstein, S. (1995). Knowledge-based anytime computation. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 775–781, Montréal, Québec, Canada. Morgan Kaufmann.

[Myers, 1999] Myers, K. L. (1999). CPEF: A continuous planning and execution framework. *AI Magazine*, 20(4):63–70.

[Nareyek, 1999] Nareyek, A. (1999). Applying local search to structural constraint satisfaction. In *Proceedings of the 1999 IJCAI Workshop on Intelligent Workflow and Process Management: The New Frontier for AI in Business*, Stockholm, Sweden.

[Nareyek, 2000] Nareyek, A. (2000). Open world planning as SCSP. In *Papers from the AAAI-2000 Workshop on Constraints and AI Planning*, pages 35–46, Austin, Texas, USA. AAAI Press.

[Nareyek, 2001] Nareyek, A. (2001). Using global constraints for local search. In *Constraint Programming and Large Scale Discrete Optimization*, volume 57 of *DIMACS*, pages 9–28. American Mathematical Society Publications.

[Nareyek, 2002] Nareyek, A. (2002). Intelligent agents for computer games. In *Computers and Games, Second International Conference (CG 2000)*, volume 2063 of *Lecture Notes in Computer Science*, pages 414–422. Springer.

[Nareyek and Sandholm, 2003] Nareyek, A. and Sandholm, T. (2003). Planning in dynamic worlds: More than external events. In *IJCAI-03 Workshop on Agents and Automated Reasoning*, pages 30–35. Forthcoming. Available from http://www.ai-center.com/references/nareyek-03-transitions.html.

[Nau et al., 1999] Nau, D., Cao, Y., Lotem, A., and Muqoz-Avila, H. (1999). SHOP: Simple hierarchical ordered planner. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI 99)*, pages 968–973, Stockholm, Sweden. Morgan Kaufmann.

[Newell and Simon, 1972] Newell, A. and Simon, H. (1972). *Human Problem Solving*. Prentice Hall.

[Newell and Simon, 1969] Newell, A. and Simon, H. A. (1969). GPS, a program that simulates human thought. In Feigenbaum, E. A. and Feldman, J., editors, *Computers and Thought*, pages 279–293. R. Oldenbourg KG. Reprinted in Allen J., Hendler J. and Tate A., editors. *Readings in Planning*, pages 59-66, 1990. Morgan Kaufmann.

[Nilsson, 1994] Nilsson, N. J. (1994). Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158.

[Nilsson, 1998] Nilsson, N. J. (1998). *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann.

[Pearl, 1984] Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

[Penberthy and Weld, 1992] Penberthy, J. S. and Weld, D. S. (1992). UCPOP: A sound, complete, partial order planner for ADL. In Nebel, B., Rich, C., and Swartout, W., editors, *KR'92. Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, pages 103–114. Morgan Kaufmann.

[Pollack and Horty, 1999] Pollack, M. E. and Horty, J. F. (1999). There's more to life than making plans. *AI Magazine*, 20(4):71–83.

[Poole, 2000] Poole, S. (2000). *Trigger Happy: The Inner Life Of Videogames*. Fourth Estate.

[Prendinger and Ishizuka, 1998] Prendinger, H. and Ishizuka, M. (1998). APS, a Prolog-based anytime planning system. In *Proceedings 11th International Conference on Applications of Prolog (INAP-98)*, pages 144 – 149, Tokyo, Japan.

[Pryor and Collins, 1993] Pryor, L. and Collins, G. (1993). Cassandra: Planning for contingencies. Technical report, The Institute for the Learning Sciences, Northwestern University. Technical Report No. 41.

[Russell and Norvig, 1995] Russell, S. J. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice-Hall International, Inc.

[Sacerdoti, 1974] Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135. Reprinted in Allen J., Hendler J. and Tate A., editors. *Readings in Planning*, pages 98-108, 1990. Morgan Kaufmann.

[Scheutz and Logan, 2001] Scheutz, M. and Logan, B. (2001). Affective vs. deliberative agent control. In *Proceedings of the AISB'01 Symposium on Emotion, Cognition and Affective Computing*, pages 1–10, York, UK.

[Schoppers, 1987] Schoppers, M. J. (1987). Universal plans for reactive robots in unpredictable environments. In McDermott, J., editor, *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, pages 1039–1046, Milan, Italy. Morgan Kaufmann.

[Shapiro, 1999] Shapiro, D. (1999). Controlling gaming agents via reactive programs. In *Papers from the 1999 AAAI Spring Symposium on Artificial Intelligence and Computer Games*, pages 73–76.

[Shoaff et al., 1994] Shoaff, W., Thiebaux, S., Hertzberg, J., and Schneider, M. (1994). A stochastic model of actions and plans for anytime planning under uncertainty. In Backstrom, C. and Sandewall, E., editors, *Current Trends in AI Planning*, pages 292–305. IOS Press.

[Sloman, 1978] Sloman, A. (1978). *The Computer Revolution in Philosophy*. Harvester Press (and Humanities Press), Hassocks, Sussex. Available from http://www.cs.bham.ac.uk/research/cogaff/crp.

[Sloman, 1993a] Sloman, A. (1993a). The mind as a control system. In Hookway, C. and Peterson, D., editors, *Philosophy and the Cognitive Sciences*, pages 69–110. Cambridge University Press.

[Sloman, 1993b] Sloman, A. (1993b). Prospects for AI as the general science of intelligence. In Sloman, A., Hogg, D., Humphreys, G., Partridge, D., and Ramsay, A., editors, *Prospects for Artificial Intelligence*, pages 1–10. IOS Press, Amsterdam.

[Sloman, 1994] Sloman, A. (1994). How to design a visual system – Gibson remembered. In D.Vernon, editor, *Computer Vision: Craft, Engineering and Science*. Springer Verlag.

[Sloman, 1995] Sloman, A. (1995). Exploring design space and niche space. In *Proceedings of the 5th Scandinavian Conference on AI*, Trondheim, Norway. IOS Press.

[Sloman, 1998] Sloman, A. (1998). The "semantics" of evolution: Trajectories and trade-offs in design space and niche space. In Coelho, H., editor, *Progress in Artificial Intelligence, 6th Iberoamerican Conference on AI (IBERAMIA)*, pages 27–38. Springer, Lisbon, Portugal.

[Sloman, 2002] Sloman, A. (2002). Architecture-based conceptions of mind. In *In the Scope of Logic, Methodology, and Philosophy of Science (Vol II)*, volume 316 of *Synthese Library*, pages 403–427. Kluwer.

[Sloman and Logan, 1999] Sloman, A. and Logan, B. (1999). Building cognitively rich agents using the sim_agent toolkit. *Communications of the ACM*, 43(2):71–77.

[Sloman and Logan, 2000] Sloman, A. and Logan, B. (2000). Evolvable architectures for human-like minds. In Hatano, G., Okada, N., and Tanabe, H., editors, *Affective Minds*, pages 169–181. Elsevier.

[Sloman and Scheutz, 2002] Sloman, A. and Scheutz, M. (2002). A framework for comparing agent architectures. In *UKCI'02: UK Workshop on Computational Intelligence*, Birmingham.

[Tate, 1977] Tate, A. (1977). Generating project networks. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI-77)*, pages 888–893, Cambridge, MA, USA. William Kaufmann. Reprinted in Allen J., Hendler J. and Tate A., editors. *Readings in Planning*, pages 291-296, 1990. Morgan Kaufmann.

[ten Teije and van Harmelen, 2000] ten Teije, A. and van Harmelen, F. (2000). Describing problem solving methods using anytime performance profiles. In *ECAI 2000, Proceedings of the 14th European Conference on Artificial Intelligence*, pages 181–186, Berlin, Germany. IOS Press.

[Valve, 1998] Valve (1998). *Half-Life*. Sierra Studios.

[Valve, 2000] Valve (2000). *Counter-Strike*. Sierra Studios.

[Wilkins, 1988] Wilkins, D. E. (1988). *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann Publishers Inc.

[Wilkins et al., 1995] Wilkins, D. E., Myers, K. L., Lowrance, J. D., and Wesley, L. P. (1995). Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI*, 7(1):197–227.

[Winograd, 1972] Winograd, T. (1972). Procedures as a Representation for Data in a Computer Program for Understanding Natural Language. *Cognitive Psychology*, 3(1). (Later published as a book *Understanding Natural Language*, Academic Press, 1972).

[Wood, 1993] Wood, S. (1993). *Planning and Decision Making in Dynamic Domains*. Ellis Horwood.

[Wooldridge and Jennings, 1995] Wooldridge, M. and Jennings, N. (1995). Intelligent agents: theory and practice. *Knowledge Engineering Review*, 10(2).

[Wright, 1997] Wright, I. (1997). *Emotional Agents*. PhD thesis, School of Computer Science, The University of Birmingham.

[Wright and Marshall, 2000] Wright, I. and Marshall, J. (2000). Egocentric AI processing for computer entertainment: A real-time process manager for games. In *Game-On 2000, 1st International Conference on Intelligent Games and Simulation*, pages 42–46, London, UK.

[Wright et al., 1996] Wright, I., Sloman, A., and Beaudoin, L. (1996). Towards a design-based analysis of emotional episodes. *Philosophy Psychiatry and Psychology*, 3(2):101–126. Reprinted. in R.L.Chrisley (Ed.), *Artificial Intelligence: Critical Concepts in Cognitive Science*, Vol IV. Routledge, 2000.

[Yang et al., 1991] Yang, Q., Tenenberg, J. D., and Woods, S. (1991). Abstraction in nonlinear planning. Technical report, University of Waterloo, School of Computer Science. Technical Report CS-91-65.

[Zilberstein, 1996] Zilberstein, S. (1996). Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83.

[Zilberstein and Russell, 1993] Zilberstein, S. and Russell, S. J. (1993). Anytime sensing, planning and action: A practical method for robot control. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 1402–1407, Chambéry, France. Morgan Kaufmann.

[Zilberstein and Russell, 1996] Zilberstein, S. and Russell, S. J. (1996). Optimal composition of real-time systems. *Artificial Intelligence*, 82(1-2):181–213.