Reasoning about Goal-Plan Trees in Autonomous Agents: Development of Petri net and Constraint-Based Approaches with Resulting Performance Comparisons

SHAW, PATRICIA,H

Reasoning about Goal-Plan Trees in Autonomous Agents:

Development of Petri net and Constraint-Based Approaches with Resulting Performance Comparisons

A thesis submitted to the University of Durham
in partial fulfilment for the degree of
Doctor of Philosophy

2010

By
Patricia H. Shaw
School of Engineering and Computing Sciences

Abstract

Multi-agent systems and autonomous agents are becoming increasingly important in current computing technology. In many applications, the agents are often asked to achieve multiple goals individually or within teams where the distribution of these goals may be negotiated among the agents. It is expected that agents should be capable of working towards achieving all its currently adopted goals concurrently. However, in doing so, the goals can interact both constructively and destructively with each other, so a rational agent must be able to reason about these interactions and any other constraints that may be imposed on them, such as the limited availability of resources that could affect their ability to achieve all adopted goals when pursuing them concurrently. Currently, agent development languages require the developer to manually identify and handle these circumstances.

In this thesis, we develop two approaches for reasoning about the interactions between the goals of an individual agent. The first of these employs Petri nets to represent and reason about the goals, while the second uses constraint satisfaction techniques to find efficient ways of achieving the goals. Three types of reasoning are incorporated into these models: reasoning about consumable resources where the availability of the resources is limited; the constructive interaction of goals whereby a single plan can be used to achieve multiple goals; and the interleaving of steps for achieving different goals that could cause one or more goals to fail.

Experimental evaluation of the two approaches under various different circumstances highlights the benefits of the reasoning developed here whilst also identifying areas where one approach provides better results than the other. This can then be applied to suggest the underlying technique used to implement the reasoning that the agent may want to employ based on the goals it has been assigned.

Declaration

No part of the material presented in this thesis has previously been submitted by the author in support of an application for another degree or qualification of this or any other university or other institute of learning. All the work presented here is the sole work of the author and no one else.
This research has been documented, in part, within the following publications:

- Shaw, P. and Bordini, R. 2007. Towards alternative approaches to reasoning about goals. In Proc. 5th Int. Workshop on Declarative Agent Languages and Technologies. Springer.

- Shaw, P., Farwer, B. and Bordini, R. H. 2008. Theoretical and experimental results on the goal-plan tree problem. In Proc. 7th Int. Conf. on Autonomous Agents and Multiagent Systems.

Contents

List of Tables

List of Figures

Acknowledgements

I would firstly like to thank my supervisor, Dr. Rafael Bordini, who inspired me as an undergraduate student to pursue research in the field of agents and to whom I owe a debt of gratitude for all his guidance and the encouragement he has provided throughout course of this research.

Also from the agents research group, I would like to thank Dr. Berndt Farwer for all his help with Petri nets and his encouragement. Thanks also go to all my other colleagues, for their moral support and encouragement to keep persevering despite the problems encountered along the way. I would also like to thank all the people who have inspired me to undertake research and work towards this PhD.

My thanks also go to all the people I have met at conferences and provided the opportunity to discuss my ideas, especially John Thangarajah for the discussions we have had regarding my ideas for potential alternative approaches to improve the reasoning of agents.

I would like to thank all my friends for their inspirational and morale-boosting talks to keep me going and for all the help and support they have provided along the way. In particular, I would like to send special thanks to Elly for all the advice she has given me, to Tally for helping me to find a needle in a haystack, and Chris for keeping me in touch with life outside academia.

A very special thank you goes to my parents for their love and support throughout, with anything from proof reading to supplying the chocolate to keep me going. I am very grateful for all the guidance and encouragement they have provided over the years to get me to this point. In addition, I would like to thank my wonderful dog, Lucy, for all the long walks she has taken me on to think over the research and without whom I may have gone insane over the last few years.

Finally, many thanks go Professor Malcolm Munro for offering me the DTA

funding from the Engineering and Physical Sciences Research Council (EPSRC) giving me the opportunity to undertake this research, along with Professor Dave Robertson and Dr. Magnus Bordewich for their insights and suggestions for continuing with this research. Thanks also go to Dr. Nick Holliman for giving me access to the array of computers that allowed me to complete all my experiments in just one month, compared to the 6 months it would have otherwise taken.

Chapter 1

Introduction

Agent technology is a growing area of research and an increasingly popular methodology for implementing systems in industry as well as academia [Alshamsi *et al.*, 2009, Mora *et al.*, 2008, Ceccaroni and Robertson, 2000]. Agents are used by space agencies to aid in the control of deep space probes [Muscettola *et al.*, 1998, Truszkowski *et al.*, 2006], as well as in many applications closer to home [Tsai *et al.*, 2009, Jakob *et al.*, 2008, Paruchuri *et al.*, 2006]. They are becoming increasingly used as personal assistants [Varakantham *et al.*, 2005] for anything from trading to finding information leading to controlling autonomous robots for transport or surgery [Palmer, 2009], along with helping the coordination of disaster response teams [Tambe *et al.*, 2005, Schurr *et al.*, 2005, Nourbakhsh *et al.*, 2005, The RoboCup Federation, 2009a].

Agents are defined as being situated in an environment, autonomous, reactive, proactive, flexible and social. In order for agents to be able to satisfy this definition they need to reason about the environment they are situated in and how to devise a method of achieving their objectives through acting on the environment to change it. When defining agents, the developers provide a wide selection of plans from which the agent can select the most suitable depending on the situation. This leads to the main advantage of intelligent agents being able to operate autonomously and achieve their goals in highly dynamic environments.

A popular architecture used by developers for defining agents is based on a philosophy of human intentions using Beliefs, Desires and Intentions (BDI) [Rao

and Georgeff, 1995, Wooldridge and Jennings, 1995]. The agent has a set of beliefs about the environment, its goals and about any other agents with whom it may be interacting with. Their desires represent the states within the environment they would like to achieve, or the goals they would like to achieve. However it is often not possible to achieve all of these so the agent chooses a subset to commit to. This subset forms the intentions which the agent is committed to achieving.

Many applications require the agents to achieve multiple goals and often in parallel, therefore it is in the commitment to achieving certain goals, followed by the autonomous selection of plans used to achieve the goals that can cause the agent considerable difficulties. When committing to goals the agent needs to consider whether the goals are compatible, such that all committed goals can be achieved concurrently. Once goals have been committed to then the selection of plans to achieve the goals and the order in which the selected plans are executed can also have an impact on the achievability of other goals. The main aim of this thesis is to investigate approaches for reasoning about goals that can be practically incorporated into agents. This is to improve the efficiency and effectiveness of the agents, specifically their ability to reason about which goals they can safely commit to achieving and how to safely go about achieving the goals they have committed to.

Within agents, goals can be split into a variety of different categories. Two categories that are commonly used for describing goals are "Achievement goals" and "Maintenance goals". As the names suggest, the achievement goals aim at achieving a desired state within the environment, while the maintenance goals are concerned with preserving a state in the environment. The driver of a vehicle could be given the achievement goal of safely transporting the passengers to their desired location, while having the maintenance goal of maintaining the speed at the safe and legal limit for the road they are on.

Limited resource availability Many application areas are constrained by the availability of resources in some form or another. This could be simply disk storage space, memory and processor availability for computations, or it could refer to more physical resources such as the fuel used to power a vehicle or the money to purchase products. Some of these can be considered as *reusable* resources, such

as memory and processors that become available again when the computation has finished, while others can be considered to be *consumable* such as the petrol in a vehicle. Once the fuel has been used, it cannot be reclaimed, so more needs to be purchased.

These same resource restrictions affect agents, so an intelligent agent needs to take them into consideration when reasoning about the goals to which it can commit and the plans it uses to achieve them. If the goals the agent is considering committing to require some resources that will be consumed and the availability of these resources is limited then the agent may not be able to achieve all the goals. It would therefore be irrational for an agent to commit to achieving them, as attempting to achieve them concurrently could cause all the goals to fail when the resources run out. For example, if an agent has 100 units of energy and one goal requiring 80 units of energy and a second goal requiring 60 units, it would not be feasible for the agent to achieve both goals. If the agent attempted to pursue both goals in parallel, the most likely outcome is that the agent will run out of energy before either of the goals has been achieved.

As the agent has a range of possible plans that can be used to achieve a given goal, it is also possible that each of these plans will have different resource requirements. If the agent commits to achieving the goal requiring at least 60 units and is asked to achieve another goal requiring at least 40 units, then provided the agent is careful, it should be able to achieve both goals. However, if that agent wastes resources by selecting plans with higher resource requirements than necessary then one or both goals could still fail when the resources run out.

Positive goal interaction Within an individual agent it is possible for some of its goals to have common properties or to produce some similar effects. For example, if an agent is given the two separate goals of buying a shirt and a tie from a shop, where the shop is the same, it makes sense for a rational agent to make just one trip to the shop and purchase both items at the same time [Horty and Pollack, 2004]. Another example of an application involving a single agent is that of a Mars rover agent [Washington *et al.*, 1999]. A simplified version of the rover will be given a variety of goals involving taking a selection of samples at different locations and transmitting the results back to Earth via a base station at

the landing site. While the batteries are rechargeable, the energy can be considered consumable between charging cycles.

In the Mars rover example, if the agent is given the goals of taking rock and soil samples at the same location, they can either move to the location, take the first sample, return to the base station to transmit the results, go back to the same location to take the second sample and finally return to base station again to transmit the second results, or the agent can take both samples at the same time, and transmit the results together.

Typically each of the goals will require the execution of multiple plans in order to achieve it and while the first approach for the Mars rover achieves the two goals sequentially, the second approach is able to interleave the plans in such a way as to be beneficial to both goals being pursued concurrently. It should also be clear that through the positive interleaving of plan executions, resources can also be saved, such as the energy required for a second trip not being needed. This can allow an agent to achieve more goals despite the limited availability of resources.

Negative goal interaction While it is possible to interleave the plans of two or more goals in such a way as to benefit each of the goals concerned, it is equally possible for poor interleaving of plans to have the opposite effect. This can result in effects that had been achieved by one plan, that were needed by a later plan, being undone by the plans of another goal. Under extreme circumstances, this could cause one or more goals to fail. An example of this negative interleaving is where a Mars rover has two goals, each taking a soil sample at a different location. The rover will have a selection of plans for moving to the two locations and taking the soil samples. If the agent executes the movement plan for the first goal taking it to location A then executes the movement plan for the second goal before the soil sample plan for the first goal, this will either cause the first goal to fail, or require it to waste resources returning to the first location again to get the first sample. In the worse case, an irrational agent could be caught going back and forth between the two locations until it ran out of energy, never taking either sample. Clearly this needs to be avoided, and the agent given the ability to reason about when there is a risk of this occurring in order to avoid it.

Summary of contributions This thesis considers the problems related to reasoning about achievement goals within a single agent when taking into account the limited availability of resources, along with the potential for positive and negative interactions between goals when attempting to achieve them concurrently. Two new approaches have been defined here for modelling this reasoning, these being, *i*. A Petri net based model (see chapter 4) and *ii*. Constraint satisfaction based model (see chapter 5).

This research follows on from the work of Thangarajah *et al.* [2002, 2003a,b], Thangarajah and Padgham [2004], Thangarajah [2004] where mechanisms for performing the three types of reasoning discussed above were developed. They define a *Goal-Plan tree* structure for representing the goals and the plans that can be used to achieve them. These plans may themselves contain *subgoals* with further plans to achieve them forming a tree structure.

The reasoning approach developed by them generates large amounts of summarised information for identifying where interactions are likely to occur between the goals, be they positive or negative interactions, along with details of the resources required by each of the goals. This summary information consists of separate lists of potentially and definitely interacting plans for positive and negative interactions along with lists of resources that will definitely be required and some that may be required. Their approach is discussed in more detail in chapter 3 with a comparison performed between the approaches developed here and their approach presented in chapter 7.

While the approach by Thangarajah *et al.* is based on the use of summary information, the approaches developed here look at where it is possible to reduce or even remove entirely the dependence on this summary information, without losing any of the improvements in efficiency and effectiveness that they have provided where possible. An experimental analysis of the outcomes of the new approaches developed here is presented in chapter 6. This evaluation considers three abstract scenarios based on different goal-plan tree structures, analysing the benefits of each of the types of reasoning performed both independently and in conjunction with the other types, while identifying situations where each of the two different approaches maybe better suited over the other.

Models As stated earlier, two different models are used in this thesis to represent the reasoning about goals within an individual agent. These models use Petri nets and constraint satisfaction techniques to describe and reason about the problem.

Petri nets are mathematical models, with an intuitive diagrammatic representation, used for describing and studying concurrent systems [Peterson, 1981]. They can represent diagrammatically the flow of control through systems, along with the movements of resources that can be consumed. This representation of the flow of control provides a natural mapping from the goal-plan trees used to describe the problem, onto the Petri nets.

Constraint satisfaction techniques attempt to find a solution to a problem over a domain of variables that have some constraints linking the variables and restricting the possible assignment of values to variables in suitable solutions. While very different in style to the Petri nets, these also provide a natural mapping of the constraints applied by the three types of reasoning over the adoption of goals and the selection of plans for achieving the adopted goals.

The results show the benefits of the reasoning compared to the absence of any reasoning, and particularly in a broad goal-plan tree structure show the constraint-based model provides better results when reasoning about resources, while the Petri net model gives greater reductions in the number of plans used when considering positive interactions. By combining the different types of reasoning together, even greater savings and improvements in performance can be gained as shown in chapter 6.

Outline of thesis The rest of this thesis is organised as follows:

In chapter 2, a survey of the related literature is given, detailing the concepts of agents and their goals; the types of reasoning about goals and background on the approaches used for modelling the reasoning in this thesis.

In chapter 3, the problem of reasoning about goals is explained in more detail. This covers the issues related to resources and the types of interactions between goals, identifying the similarities and differences between existing work and the work presented in this thesis, before chapters 4 and 5 describe the two approaches developed for modelling the problem and the reasoning incorporated into them.

Chapter 4 develops the first of these models using Petri nets to represent the

goal-plan trees. Each of the types of reasoning is then modelled as a series of modules that can be incorporated into the goal-plan tree and mapped onto a Petri net.

Similarly, in chapter 5, a constraint satisfaction based description is given for describing a goal-plan tree, with each of the types of reasoning formalised into predicates that can be applied as a set of constraints to the goal-plan tree model used in this approach.

In chapter 6, these two approaches are quantitatively compared to each other under a wide range of conditions within three different goal-plan tree structures to analyse their performance, and an attempt is made to identify any situations where one approach may be better suited over the other.

Finally, chapter 7 presents the conclusions drawn from this thesis, including a qualitative comparison of the approaches developed here to the approach developed by Thangarajah *et al.* A discussion is also given regarding possible areas for expansion and future research aiming to continue on from this thesis.

Chapter 2

Background

2.1 Introduction to Agents

Agent research has been formed from three main contributing research areas. These are Artificial Intelligence (AI), Object Oriented (OO) programming and Human Computer Interaction (HCI) design, with the major contribution coming from AI research, particularly the research in AI-Planning [Jennings *et al.*, 1998]. In addition to this, Distributed Computing (DC) has also provided an important basis for multi-agent systems distributed over a network.

When referring to agents, we more specifically mean software agents. A commonly used definition of an agent is that given by Wooldridge and Jennings [1995] defining an agent to be situated within an environment and autonomous. The environment can be the real world, or it may be the Internet, or simulated within a computer system, but the agent will be able to receive sensory input from the environment and its actions will endeavour to affect the environment in a particular way to help achieve its goals. These environments tend to be more complex than the sort of environment most software could be considered to be situated in. They are often more dynamic and unpredictable, and the agent may not have a complete view of its surroundings, for example in a disaster rescue scenario [Schurr *et al.*, 2005]. Often the dynamic and unpredictable aspects are brought about by the presence of multiple agents within the environment, for example robot football [Akin, 2005], but there could be other factors within the environment such as

weather that is not controlled by any agents and is simply part of the dynamic nature of the environment. These changes within the environment mean that an agent cannot always assume that its actions will be successful or that effects brought about by their actions will remain unchanged, leading to the need for agents to constantly be aware of their environment and able to respond to changes as they occur.

Autonomy means the agent should be able to act without the direct intervention of humans or other agents, and that it should have control over its own actions and internal state. This property also helps to start distinguishing agents from simply being objects as used in Object Oriented (OO) programming. Part of the autonomy requires the agent to be flexible. This is reflected by three further properties, these being reactive, pro-active, and social. The reactive property means that the agent should respond in a timely fashion to changes in the environment that it perceives. The pro-active property indicating that the agent should exhibit opportunistic and goal-directed behaviour, taking the initiative where appropriate as well as not giving up on a goal at the first failure, making the goals persistent while it is still feasible to achieve them. Finally the social behaviour expresses that the agent should be able to interact with others where necessary to aid in problem solving or achievement of goals when operating in an environment with more than one agent in it.

Standard semantics based on speech acts [Searle, 1969], have been defined to allow different agents to communicate using the same language, allowing more than the simple passing of parameters used in objects. This provides a series of performatives to 'inform', 'request' and 'agree' to queries and goals communicated between agents [FIPA, 1999]. By combining these performatives together, a very expressive language can be defined allowing the agents to communicate in detail. On receiving a message, the agent can choose how to respond to that message, for example if an agent is asked to open a door, they may or may not choose to do so, whereas an object sent the same message would automatically execute the 'open door' method that had been called to achieve the goal. In the real world, these other agents can be humans that the agent is interacting with. Together, these three properties describing flexibility encourage the agent to be more robust to change within its environment. The pro-active behaviour in particular helps to

further distinguish agents from objects as objects are more reactive when told to do something or when responding directly to an input, rather than pre-empting and making use of changes within their environment to speed up the achievement of their goals. Agents and agent oriented programming are sometimes considered to be the next step on from OO programming in the evolutionary cycle [Odell, 2002, Baldoni *et al.*, 2006, Bordini *et al.*, 2005a].

In [Tessier *et al.*, 2001], they add to this definition requiring that the agent also possesses at least a partial representation of its world. Agents that wish to be considered as intelligent should also be rational agents [Thangarajah *et al.*, 2002]. This means they should not perform any actions that negatively effect their ability to achieve their goals, whether as part of a team or individually. In order to ensure that an agent performs rationally, it needs to reason about factors such as the limited availability of any resources that are used; the actions it performs and how they interact with the effects on the environment generated by other goals; and finally, the interactions within teams and how the actions of other agents affect each others ability to achieve their given goals, including how teams could reason together to become more successful at achieving their given goals.

A commonly used agent architecture is the Belief Desire Intention (BDI) architecture, which is an example of a logic-based architecture. The different types of architectures include logic-based architectures, reactive architectures and layered architectures [Weiss, 1999], however this thesis focuses on agents developed using the BDI architecture. The BDI model is inspired by and based on a model developed by philosophers, specifically Bratman [1990], to describe human behaviour, in particular the role of intentions in practical reasoning and is a useful abstraction tool for describing complex systems [Bordini *et al.*, 2007, chapter 2]. BDI agents contain a set of beliefs in their internal state, which represents their knowledge about their environment and other agents within the system. For example, the BDI agent may believe that it is raining or that agent j is capable of a particular job. The beliefs of an agent are updated from its perceptions of its environment and any interactions it has with other agents. The percepts received by the agent allow them to learn about the environment in which they are situated by receiving sensory inputs from the environment or through the outcomes of sensory actions such as touching something. The agent may not be able to perceive the whole

environment at any one time, so the agent needs to record what it has learnt as beliefs about the environment. These percepts then form the basis of the agent's belief base on which an agent can start to make decisions about actions to perform, whilst keeping in mind that the environment could possibly change whilst the agent is making its decisions, thereby falsifying the beliefs on which the decision is based. As a result, the agent needs to take into consideration the rate of change within the environment when making decisions and where necessary avoid spending longer than necessary on its deliberation.

The desires or goals represent what the agent would like to achieve, while the intentions represent a set of plans and actions the agent has committed to perform [Georgeff *et al.*, 1999]. It is possible that some of the desires may conflict, so a rational agent should only commit to achieving the goals that it can actually achieve. The agent uses a plan library containing generic plans in order to aid its planning and achieve the goals to which it has committed. The plans are partially instantiated when the agent is initialised, with pre-conditions restricting when it is appropriate to use it. The agent is then able to make a choice of which plans it can use in order to achieve its goals. The agent leaves the actual commitment to specific plans as late as possible to allow it to consider any updates to its beliefs and any last-minute changes within the environment as they occur.

In [Georgeff and Lansky, 1986], they present the Procedural Reasoning System (PRS), one of the first implementations of an agent-oriented system based on the BDI architecture. The system provides a library of plans instead of attempting to generate plans. This means that the agent is able to be more reactive to the changes and identify sequences of plans, rather than planning the individual actions.

A plan consists of a sequence of actions, possibly including further subgoals that will also need to be achieved in order for the plan to be successfully achieved. Actions are often considered to be atomic and often instantaneous, although duration can be applied to them, with partial effects resulting, such as only making it part way to a desired destination. They are performed by the agents actuators and represent the agent's attempt to make changes to the environment, for example the agent changing location, or attempting to move an object within the environment.

An example programming language for implementing BDI agents is AgentSpeak [Rao, 1996, Bordini *et al.*, 2002] that is used with Jason, a Java-based platform for developing multi-agent systems and used as an interpreter for AgentSpeak [Bordini *et al.*, 2007]. Together they provide a formally defined method of producing plan libraries for agents to draw from, and formally defined semantics for communication. Jason is an interpreter for AgentSpeak that provides a Java based definition for environments giving agents perceptions to interact with the environment, along with a facility to distribute multi-agent systems over a network [Bordini *et al.*, 2005b]. Other languages based on the BDI architecture model include PRS [Georgeff and Lansky, 1986], dMARS [D'Inverno *et al.*, 2004], 2APL [Dastani, 2008], JAM [Huber, 1999] and JACK [Busetta *et al.*, 1999] amongst others [Bordini *et al.*, 2005a, 2009].

While most agent research is done by simulating small test problems, there are already many typical applications where agents are being used. Some of these applications for agents are as follows:

- Distributed sensor net where agents represent the sensors and coordinate with neighbours to track targets moving through the net. In [Nair *et al.*, 2005], the sensors are laid out in a grid and to ensure a target is monitored and its location accurately recorded there needs to be at least three sensors monitoring it around the square that the target currently occupies. In this example, sensors can only detect objects in one direction; however they can also rotate to view in other directions.

- Personal assistants interacting with humans and each other within an organisation to aid meeting organisation and time management [Scerri *et al.*, 2002].

- Air traffic control where agents represent the aircrafts, planning flight paths in accordance with a set of constraints such as minimum distance allowed between two aircrafts and minimum fuel consumption on route to destination. Where two agents have flight plans that come too close to each other the agents must resolve the conflict [Jennings *et al.*, 1998].

- Transportation system with a car sharing application representing the people

who are able to offer transport to various locations and the people who would like the transport [Jennings *et al.*, 1998].

- NASA's deep space probes where autonomy is a requirement for the ability to recover from failure in a highly unpredictable environment [Muscettola *et al.*, 1998, Truszkowski *et al.*, 2006].

2.2 Reasoning about Goals

There are multiple types of conflicts that rational agents need to be aware of; these can be internal to the individual agent, or external between two or more agents [Hannebauer, 2001]. While conflicts can occur in social interactions, when attempting to delegate or collaborate over a set of given tasks [Castelfranchi and Falcone, 2001], the main focus of this thesis is to look at conflicts between goals within an individual agent.

Tessier *et al.* [2001] classifies conflicts into two key categories: *Physical* and *Knowledge*. The physical conflicts are factors such as resources or space, while the knowledge conflicts refer to differences in opinions or points of view between the agents. It is the former category of conflict that is the main interest in this thesis. In [Fisher and Ghidini, 2009] they examine the concept of resource and space bounded agents, however the bounding is based on the depth of nested beliefs held by the agent and the length of time spent reasoning about which action to perform next, while the reasoning in this thesis focuses on the limited availability of resources consumed by the actions and the interactions between different goals the agent has.

The conflicts can arise within a single agent when it has taken on two or more goals that are not entirely compatible and the agent is attempting to achieve them both concurrently [Hannebauer, 2001]. They may be caused if there is a limited amount of consumable resources available [Thangarajah *et al.*, 2002, Raja and Lesser, 2004a] such that there is insufficient resources available to achieve all the goals, or it may be due to the effects the actions involved in achieving the goals have on the environment between two concurrent goals [Thangarajah *et al.*, 2003a,b].

In [Winikoff *et al.*, 2002] formal definitions for modelling declarative goals are given, considering the goal as two parts; a declarative description of the state sought and a procedural set of plans for achieving the desired state. They require the goals for rational agent to be persistent, unachieved, possible, consistent and known. This means the agent should know about all the goals it has and must continue to attempt to achieve a given goal while it is unachieved and still possible to achieve. The final condition they define for the rational agent is that of consistency, stating that the agent should not attempt to pursue conflicting goals simultaneously. In order for this condition to be met however, either the agent programmer needs to be careful to ensure the agent is only given consistent goals, and never accepts any other goals from other agents that may conflict or, as is more suited to the agent paradigm, the agent needs to be given the ability to reason about its own goals and whether it is safe to attempt to achieve new goals in parallel. To this end, in their approach to reasoning about goals, they start by defining a formal semantics and the operations using the semantics for defining interactions between goals. Using these it is then possible to perform reasoning on goals separately from plans, so you can reason if a goal has become impossible and should be dropped, or if a goal has been achieved before the plans have finished executing. This has provided the basis on which the same researchers have proposed a particular set of approaches for such reasoning on goals under different conflicts and interactions [Thangarajah *et al.*, 2002, 2003a,b]. In the first of these three papers they describe a formal method for reasoning under the limited availability of consumable and reusable resources, while in the other two papers they reason about the effects of actions and how they interfere with other actions for achieving multiple concurrent goals. This interference between effects caused can either be negative or positive depending on whether the action hinders or aids other goals in their completion. Each of these types of reasoning are discussed in more detail in the following sections.

In each of the three papers by Thangarajah *et al.* a goal-plan tree (see figure 2.1) is used to represent the structure of the various plans and subgoals related to each goal. In order for a plan to be completed within the tree, all of its subgoals must first be completed, however to achieve a goal or subgoal only one of its possible alternative plans needs to be achieved. At each node on the tree, “summary

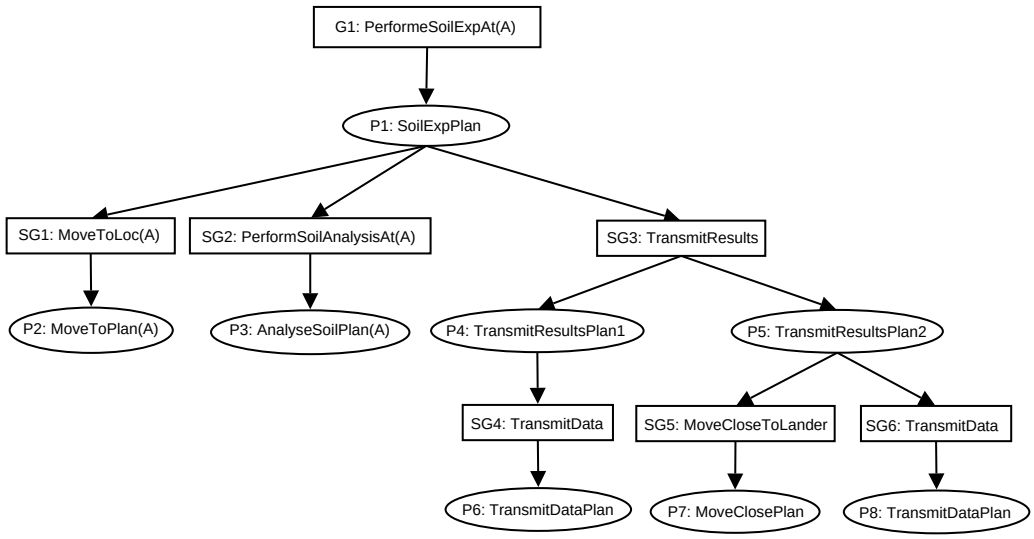


Figure 2.1: Goal-plan tree for a Mars rover as used by Thangarajah *et al.* The goals and subgoals are represented by rectangles while the plans are represented by ovals

information" is used to represent the various constraints under consideration. The reasoning done is solely internal to the individual agent. The goal-plan tree shown in 2.1 represents a single top-level goal given to a Mars rover to extract a soil sample from a given location and transmit the results back to Earth via the base station at the landing site as used by Thangarajah *et al.*

2.2.1 Reasoning about Resources

When referring to resources, Thangarajah *et al.* [2002] are referring to both reusable and consumable resources; for example a communication channel is a reusable resource while energy or time is consumed so they cannot be reused. Summaries of the requirements for these are passed up the tree towards the goal, deriving what resources are *necessary* in order to achieve the goals, and also what resources may *potentially* be used. A necessary resource is defined as one that will always be used regardless of the selection of plans when there is a choice between multiple plans, while a potential resource is one that is only used by some of the plans that can be selected, so it may not be needed, in certain circumstances. They introduce a notation, based on set theory, allowing the derivation of summaries

for the resource requirements of each (sub)goal and plan. These can then be used to reason about where conflict may occur so that it can be avoided by choosing suitable alternative plans or appropriately ordering plan execution. An algorithm is also given to compute how safe it is to add on a new goal to the existing set of goals. It does this by checking each type of resource used, looking to see if adding the new goal would cause any conflict in any of them. If it would introduce conflict, a check is also performed to see if it can be scheduled to avoid the conflict where the resources involved are reusable. The initial formation of the goal-plan tree and summary information for the agent is produced at compile time, and the highlighted conflicts are then monitored at runtime in an attempt to avoid conflict.

Empirical results from experiments done using this reasoning are given in [Thangarajah and Padgham, 2004]. They consider goal-plan trees of depth 2 and depth 5, varying the amount of parallelism between multiple goals, and the amount of competition for the resources either by reducing the availability or increasing the number of goals vying for the same resource. The reasoning is implemented as an extension to the JACK agent development system [Busetta *et al.*, 1999] and called X-JACK. The performance of X-JACK is compared against the performance of JACK (i.e., without any of the additional reasoning), and shows an improvement in performance regarding the number of goals successfully achieved, with only a half second time increase in the computation cost.

In comparison, Raja and Lesser [2004a] also consider a single agent's limited resources when deliberating and performing actions in a multi-agent environment, where coordination and negotiation with the other agents is required. This work is based on their earlier work [Lesser *et al.*, 2000], where they define the *BIG agent architecture* for use on the World Wide Web to gather information while being constrained by limited resources, trading time for greater quality of results, where the longer the time taken, the greater the quality of the results returned. In their later work, they attempt to address the problem of limited resources by applying meta-level control and making use of reinforcement learning over a Markov Decision Process (MDP) to improve the performance of the agents over time.

An MDP is a mathematical framework for modelling decision-making in situations where outcomes are only partly controlled by the decision maker, with

the remainder of the outcome being random. Markov Decision Processes (MDPs) consist of an initial state, a transition model between the states and a reward function related to the current state. A solution or policy specifies what the agent should do for any state it reaches, with the expected utility of the policy being generated by the expected rewards from the event history. An optimal policy is one which yields the highest expected utility [Russell and Norvig, 2003]. Simari and Parsons [2006] considers the relationship between MDPs and the BDI architecture, mapping intentions to and from the BDI architecture onto policies in an MDP.

The approach developed by Raja and Lesser starts with a random selection of actions, from which the agents build up a set of episodes with the MDP abstracting the current state and attempting to estimate the probabilities of transitions arriving at state s' from s by taking action a. As the agent further explores its state space, the efficiency of the model increases until in the final steps of the exploration an optimal policy is expected. Each of the agents are willing to reveal information to the other agents in the multi-agent system in order to allow them all to perform better as a whole, bringing the policies of each of the agents into convergence. This leads to an increasingly large number of parameters that need to be learnt as the number of agents in the system increases, so they focused on the interactions between two agents in their experiments.

Another approach using the preferences of an agent to define an MDP in order to reason about resource allocation between self-interested agents in a multiagent system is that proposed by Dolgov and Durfee [2006]. Their approach directly links the amount of the resources to the policy, cutting out the need for utility values to be estimated, so while they also show the problem to be NP-complete, the experimental results show their approach gives large gains in computational efficiency compared to that required by a combinatorial resource-allocation approach. Alternative approaches to attempting to resolve the resource allocation have involved the use of Answer-Set Programming (ASP) [Leite *et al.*, 2009] in an attempt to solve the different Multiagent Resource Allocation (MARA) problems. These problems however are outside the scope of this thesis as we focus on reasoning about the consumable resources available to an autonomous agent working on its own in an environment.

2.2.2 Reasoning about Positive Interaction

Reasoning about the effects of actions needs to consider both positive and negative impacts in relation to other goals, and causal links that may exist between plans within a single goal. A causal link is formed where the effects of one plan are the preconditions of another, linking the two plans together. Negative interference breaks these links by changing the effects in some way between the execution of the two linked plans.

While most papers focus on the negative impacts of goals interacting with one another, some also consider the positive implications of this interaction as well. As the goals of BDI agents are modelled on human goals, examples of this reasoning can be found in philosophy literature, such as [Björnberg, 2009] where S-relations and C-relations are defined as *Supporting* or *Conflicting* relations between two or more goals. The supporting relations indicate where one goal facilitates the achievement of the other, or they facilitate the achievement of each other. Where a supporting goal is achieved, this increases the probability that the goal itself will be achieved. The author notes that when the allocation of resources is being considered, the S- and C-relations can be used to more effectively distribute resources between goals to avoid wasting the resources on goals that will be unachievable due to conflict.

Weiss [1999, section 3.5] discusses "plan merging", but this is actually referring to checking whether two plans from different agents can be executed simultaneously or whether they will interfere with each other. If they interfere, the reasoning checks to see if the conflicting plans can be scheduled to avoid the interference and if not then the plans are said to be incompatible with a knock on effect on each agent.

The positive interaction between goals has also been considered within agents that have multiple goals [Thangarajah *et al.*, 2003b, Horty and Pollack, 2004]. In the work by Thangarajah *et al.*, when reasoning about the effects caused by plans, they consider the negative and positive interactions separately. In [Thangarajah *et al.*, 2003b], they are just concerned with exploiting any positive interaction between goals. This is where two or more plans cause the same effect, so rather than executing both, it may be possible to merge the two plans, thereby improving

the performance of the agent. To represent this form of reasoning, they once again use the goal-plan tree with summary information showing the definite and potential effects of the plans and goals, with a particular method used to derive the summaries. They then describe how an agent can decide if it is feasible to merge the plans, and how to avoid waiting too long if one of the two plans for merging is reached considerably sooner than the other. It is also possible that the second plan will never be reached where the merger was only a "potential" merge, so the agent needs to be able to reason about whether it is worth waiting or not to avoid wasting too much time waiting for something that will never happen. Results from their experiments using this reasoning are presented in [Thangarajah, 2004]. These results show the expected reduction in the number of plans used compared to the same experiments being run without any reasoning. The time taken with the reasoning was actually reduced slightly due to the reduction in the number of plans being used.

Horty and Pollack also consider the merging of plans where positive interaction occurs [Horty and Pollack, 2004]. In their work, an agent evaluates the various options it has between its goals within the context of its existing plans. They use estimates for the costs of plans, and where there is some commonality between its existing plans and another plan, then the plans will be evaluated for merging. If the estimated cost of the merged plans is less than the sum of the two separate estimated costs then the plans will be merged. The example they give to illustrate this is an "important" plan of going to a shopping centre to buy a shirt, while also having a less important goal to buy a tie separately. Both plans involve getting money and travelling to a shopping centre, so if the overall cost of buying the tie at the same time as the shirt is less than that of buying the tie separately then the plans will be merged, even though the goal of having a tie is not as important. In this way, they look for the least expensive cost for the execution of the plans involved to achieve the goals.

2.2.3 Reasoning about Negative Interference

In reference to the philosophical literature on negative interaction discussed by Björnberg [2009], the interference from the C-relations that they consider is often "hard"

conflicts that completely prevent another goal from being achieved. This is often associated with a lack of a consumable resource such as money, but they also include the effects of actions from goals such as the goal to be intoxicated twenty-four hours a day and the goal to become a skilful violinist. Clearly these goals conflict with each other in such a way that one cannot be achieved alongside the other. Not all conflicts are so hard, there are "softer" conflicts that allow two conflicting goals to be taken on at the same time. Many humans and organisations will knowingly take on conflicting goals, out of which some of the conflicts will resolve themselves effectively over time, while others will come into conflict. Attempting to avoid adopting any conflicting goals will ultimately result in a system where very few or even no goals are ever adopted so very little is achieved as all actions could potentially conflict with something else. As a result, we need to find a method of handling the conflicts between the goals wherever possible.

Within agents, approaches to reasoning about negative interactions between goals include [Bonura *et al.*, 2009], which describes a development suite for BDI agent systems called *PRACTIONIST*, standing for PRACTIcal reasONIng sySTem that attempts to guarantee the consistency of intentions within a system based on the properties of goals preventing strongly inconsistent goals from being adopted at the same time, or adopting goals that cannot be achieved due to conditions currently true in the environment.

A similar approach is given by both [Pokahr *et al.*, 2005] and [Tinnemeier *et al.*, 2008]. In these types of approaches, developers performs deliberation at a goal level, deliberately abstracting away from the plans, to constrain the maximum number of parallel goals or to identify negative relationships between two goals. These relationships are defined by the developer and linked to goal templates, where each template is a goal of which multiple goal instances may be created. For example, when defining a room cleaning agent, the developer would add an "inhibits" relationships between a goal type for maintaining battery level to all the other goal types. This would then prevent the agent from attempting to achieve instances of the maintain battery goal at the same time as any other goals at run time. When the agent is deliberating about plan selection, these relationships help to speed up the deliberation cycle by reducing the number of goals and plans that need to be considered.

Meneguzzi and Luck [2007] follows on from the definitions of meta-level reasoning about goals described in [Raja and Lesser, 2004b] and applies this reasoning to consider goals before committing to them based on motivations. They do this by abstracting away from specific details about the plans or goals to produce a more generic approach to reasoning which they claim enables a more flexible specification of meta-level reasoning than that offered by more static strategies focusing on specific properties of the agents reasoning such as the negative or positive interactions and that require detailed knowledge of the underlying architecture such as those developed by Raja and Lesser [2004b], Pokahr *et al.* [2005] and Thangarajah *et al.* [2003a,b].

In the paper by Thangarajah *et al.* on reasoning about the interactions of effects, generated by the plans that are executed by the agents, they consider how to detect and avoid negative interference between goals [Thangarajah *et al.*, 2003a]. By using additional types of summaries, similar to those developed in [Clement and Durfee, 1999b], such as summaries for definite or potential pre-conditions and in-conditions along with post-conditions (i.e., effects), they monitor the causal links between effects produced by one plan which are used as pre-conditions of another to ensure these are not interfered with. The pre-conditions are conditions that need to be true before the plan can start, while the in-conditions are a subset of the pre-conditions that need to remain true for the whole duration of the plans execution. The post-conditions or effects are the expected outcomes of executing a plan. The definitions of definite and potential effects are of the same form as the necessary or possible resources used in relation to the reasoning about resources. The reasoning is done by using the summary information in a goal-plan tree, monitoring guarded sets of dependency links or causal relations. To derive the lists of interacting plans, a formal notation based on set notation is defined, to allow the agent to produce the summary information to reason about conflicting actions between its current goals and any new goals the agent may consider adopting.

When conflicts occur, scheduling can generally be used to protect these causal links until they are no longer required. Also in [Thangarajah *et al.*, 2003a], the author determines a sequence of steps for the agent to schedule and avoid interference, including checks to perform before accepting a new goal. Empirical results from

experiments using the reasoning described in this paper are given in [Thangarajah, 2004] comparing the performance of an agent with and without the reasoning, varying the same factors as they did for testing the reasoning about resources. The results show an improvement in the number of goals successfully achieved, and only slight increase in time taken to perform the additional reasoning.

While the papers by Thangarajah *et al.* have reported on experimental results for reasoning separately about each of these types of interactions between plans and goals as well as resource usage, there are no results given showing all three forms of reasoning working in conjunction. All results are given for the individual types to demonstrate the sole effects from the individual reasoning and the small amount of added computational costs associated with it, the maximum amount of computation time that was added on in any of the experiments being half a second. The lack of combined results suggests there may be the possibility of there being interference between the different forms of reasoning presented in their approach. This could be a dilemma caused between the different reasoning approaches, for example if one reasoning suggests that performing a particular plan will cause conflict while another reasoning suggests that the only other alternative will also cause conflict the agent may be unable to decide between the two without some additional overriding reasoning.

The results were also limited in the depth of trees tested. In the real world it is possible the plans and hence the goals would be slightly more complex leading to trees of greater sizes. However, as the tree grows the amount of summary information required to perform the reasoning will grow exponentially [Clement and Durfee, 2000a]. This will have a significant impact on the performance of the agent when attempting to reason about larger problems.

Prior to the time that the work by Thangarajah *et al.* was published, the Distributed Intelligent Agents Group, led by Edmund Durfee at the University of Michigan, produced some similar research for modelling and reasoning about effects, extending their work to cover Multi-agent Systems [Clement and Durfee, 1999a,b, 2000b]. In their work, they were interested in reasoning about conflicts to coordinate the actions of agents that use a system of Hierarchical Task Network (HTN) planners to coordinate their reasoning while the work by Thangarajah was based around BDI agents. In Clement and Durfee [1999b], they present the

summary information for pre-, in- and post- conditions of plans, which is adopted by Thangarajah *et al.* and used in the goal-plan trees to reason about resources and effects. The work by Clement and Durfee is discussed in more detail in section 2.3.2.

2.3 Alternative Approaches to Decision Making

In [Thangarajah *et al.*, 2002, 2003a,b] they have used a goal-plan tree with lists of summary information that are passed up the tree of plans and goals to aid the reasoning. It is based on the idea used at the University of Michigan, to form a hierarchy in the agents, through which the summary information is then transmitted to aid their planning and searching for an optimal solution to coordination and reasoning [Clement and Durfee, 1999a,b, 2000b]. While formal definitions have been produced to describe the types of reasoning under consideration, there are various other methods that could be used for modelling this type of problem, which can lead to agents using alternative approaches to reasoning about goals; the aim of the thesis is to experimentally develop two of these alternative methods, comparing their performance under varying conditions. It is predicted that each of the approaches may be better suited to certain structures of goal-plan trees over the others.

Possible alternative approaches include Petri nets [Bonnet-Torrès and Tessier, 2005], planning [Russell and Norvig, 2003, chap. 11] and CSPs [Hannebauer, 2001], amongst others, however we just consider these three here. Petri nets have previously been used to represent agents [Duvigneau *et al.*, 2003, Bonnet-Torrès and Tessier, 2005] and reasoning can be incorporated into the Petri net representations of the agents, while planning can search for the best sequence of actions for solving a problem and can be implemented using CSPs. The CSPs which can standardise the problem as a series of constraints, allow existing CSP search techniques to be used to find a solution, satisfying the given set of constraints.

2.3.1 Petri nets

Petri nets are mathematical models that can be displayed graphically for describing and studying concurrent systems [Peterson, 1981]. They consist of places and transitions that are connected by arcs, with tokens that are passed from place to place through transitions. Transitions can only 'fire' when there are sufficient tokens in each of the input places, acting as pre-conditions for the transition. The tokens are then removed from the input places, and one is placed in each of the output places. Arcs can have weights associated with them, the default weight being one. Greater weights on arcs either require the place to have at least that many tokens for the transition to fire, or the transition gives the place that number of tokens as its output (see chapter 4 for more details).

While Petri nets have not been used for this type of reasoning in agents, they have been used for related work in agents such as [Bonnet-Torrès and Tessier, 2005], which uses Petri nets to decompose team plans into plans for individual agents. They do, however, have the advantage of being mathematically formalized and have a wide range of applications [Murata, 1989, Peterson, 1981, Bakam *et al.*, 2001, Kristensen *et al.*, 1998, Leifer and Milner, 2004, Conway *et al.*, 2002]. Petri nets are also used to represent the plans used by robots [Ziparo and Iocchi, 2006, Ziparo *et al.*, 2008], for participation in RoboCup football [The RoboCup Federation, 2009b].

In [Murata, 1989], a formal definition of Petri nets is given, along with a set of constructs that can be used to model various types of functionality into a Petri net. For each of these, liveness, safety and reachability properties can be proved to ensure the correctness of the system they represent. The paper finishes by analysing algebraic or high-level Petri nets where the tokens are variables in formulas, and the weights on the arcs state the variables needed for the input and output of transitions. These can also be considered as a form of coloured Petri nets. Formal proofs have been given on the complexity of reachability on Petri nets in [Ramachandran and Kamath, 2004], showing that reachability is decidable and NP-complete.

Coloured Petri nets, are an extension to standard Petri nets where the tokens have a colour or data value associated with them and arcs have matching colours

or data values associated with the weights on the arcs. In [Cost *et al.*, 1999, 2000], coloured Petri nets are used to model conversations to provide a structure for conversations between agents, while in [Bakam *et al.*, 2001], they are used as a formal method of analysing multi-agent hunting management systems. This extension of Petri nets is also used in [Bai *et al.*, 2004] and [Weyns and Holvoet, 2002] to model multi-agent interactions and applications, such as Packet World.

Another extension to Petri nets is object oriented in style and called Reference nets or Object nets [Köhler and Rölke, 2004], where nets can be embedded within other nets. Reference nets are a development of Recursive Petri net (RPN)s that are used in [Seghrouchni and Haddad, 1996]. In that paper they use RPNs to model distributed planning and show how it can handle both positive and negative interactions between agents. This work does not use BDI agent's, instead using a definition of the agents plans, actions and methods that are also given in the paper. It does this by using the tokens in the places as pre and post-conditions for the actions which are represented by transitions in the Petri net. Planning algorithms are then used to resolve any negative or positive interactions before they can occur, to allow the actions to either be merged or be sequenced to avoid one action hindering the execution of another. Further, unlike in BDI agents, this approach does not represent goals as separate entities to the plans, rather the goals are simply the post-conditions of the plans. This means that if the goal, by some potentially unexpected change in the environment, is achieved earlier than expected then the plans still have to be executed as the completion of the goal will not have been realised.

The reference nets extension is used in [Duvigneau *et al.*, 2003] to model a multi-agent system, showing how reference nets can be effectively used for message passing as a communication mechanism.

2.3.2 Planning

Planning systems search for a sequence of steps to achieve a predefined goal state. However, in the highly dynamic environments that agents are used for, static plans will typically perform very badly. Dynamic planning offers greater flexibility, adjusting the plans each time the environment changes; however, in a dynamic

domain, the size of the search space increases rapidly, making searching intractable. Classical planning also relies on the environment being fully observable, which often is not the case in the real world [Russell and Norvig, 2003, chap.11].

In planning, each action has lists of pre-conditions and effects, leading to sequences of actions that have to be performed in order to achieve a goal. This allows plans to search either forwards or backwards to find a consistent plan, for example starting from the goal and finding actions that will satisfy the pre-conditions of the goal leading back to the current state, or starting from the current state and searching for actions whose pre-conditions are already satisfied. Often a combination of these will be used so the backwards and forwards searching meets in the middle. However, backwards searching becomes difficult if the final state cannot be explicitly defined. A consistent plan is one without any conflict in it such as those that were covered in section 2.2.3 on negative interactions between goals. Conflicts can occur when actions undo the desired effects produced by other actions, or interrupts causal links between two actions. A consistent plan produced by a planner will be a sequence of actions without conflicts and also requires that there are no cycles in the constraints that would cause infinite loops or deadlock if an agent attempted to execute them.

Planning agents can operate in one of two ways, either by producing a complete plan resolving any conflict between actions before it starts to execute any actions, or it can execute a given plan irrespective of any unresolved conflicts and simply re-plan when the plan fails. Heuristics can be derived in relation to the structure of the problem to aid the searching of suitable plans, and partial planning can be done where the goal can effectively be split up into subgoals. Job shop scheduling is a category of time related planning problems, with a lot of research focusing on improving the efficiency and solving ability of algorithms for real world planning [Sutton and Barto, 1998, chap. 9 & 11][Kumar and Rajotia, 2006, Teo *et al.*, 2005].

Early work in the use of planning in multi-agent systems was done by Georgeff [1983] where he used planning to synthesise multi-agent plans from single agent plans. This was done by identifying where communication needed to be inserted in order for the different agents to synchronise their actions and avoid conflicts.

Industrial applications of planners have covered many areas including waste

water treatment plants where a reactive planner *WaRP* was developed to be capable of actively adapting to changes in its environment through the use of incremental planning. This allowed them to keep the planning time to a minimum so that if changes did occur the system could react quickly to them [Ceccaroni and Robertson, 2000].

While in planning they sometimes talk about subgoals and their plans [Chen *et al.*, 2006] these are produced by the planner attempting to "divide and conquer" the problem it is solving. By partitioning the problem into smaller sub-problems that are similar to the main problem, it is possible to reuse the same planner and heuristics to resolve the smaller problem then pass this up to the next level. This is a similar idea to that used in another major branch of planning, which uses hierarchical decomposition to form Hierarchical Task Network Planning [Erol *et al.*, 1994]. In this approach, the initial view of the problem is very abstract, for example the initial plan may simply be to build a house. Each action can then be decomposed into further plans with associated subgoals, forming a decomposition hierarchy until the primitive actions are reached at the bottom [Russell and Norvig, 2003, chap. 12].

This form of planning is used by Clement and Durfee [1999a] and Clement *et al.* [2002] to model reasoning about conflicts between agents. They cut computation costs by reasoning about plans at higher levels of abstraction; however this also restricts the flexibility of the planning. They use the concept of summary information as defined in [Clement and Durfee, 1999b] to identify when conflicts may occur between two or more agents. Included in the summary information are details regarding the pre and post-conditions standard to planning, but also included are in-conditions defining constraints that must also be true while the plan is being executed. The algorithm for the reasoning and evaluation of its performance is then detailed in [Clement and Durfee, 2000b], looking at both the computational complexity of the algorithm and at experimental results comparing CPU time used by their new algorithm with summary information to a similar existing algorithm, Fewest Alternative First, without the use of the summary information. The results show the original algorithm sharply increasing in duration over the first 5 problems before running out of memory and so being unable to complete any further problems, while the new algorithm slowly increased its duration as the number of

problems to solve increased. The experiments stop when the new algorithm also runs out of memory.

The results show the benefits of performing the reasoning at higher levels of abstraction with the summary information allowing reasoning to determine where plans can be merged, or where conflict may occur that needs to be avoided by ordering (i.e., scheduling) the plans accordingly. However, if solutions cannot be found at high levels of abstraction then the computational cost of maintaining the summary information increases with the number of branches at the different levels of abstraction, growing exponentially, as in the case of the goal-plan tree used by Thangarajah *et al.*

In [Sardina *et al.*, 2006], they look at how Hierarchical Task Network (HTN) planning can be integrated into BDI agents. By controlling how much planning is done and what information is used, reusing as much as possible from the BDI program, they give the BDI agent the ability look ahead in a static environment before committing to any plans. In [Walczak *et al.*, 2007] they also consider how planning can be introduced into BDI architectures, including how to handle plan failures as the planner could potentially produce an infinite number of plans for the BDI agent to try in an attempt to achieve its goal. Here the BDI reasoning needs to consider four possibilities: firstly the planner cannot find any solution at the current time, in which case the agent may wait for other external changes to take place before asking for another plan; secondly, the planner may only be able to offer a partial solution which could lead to a dead end or open more possibilities; thirdly the planner has run out of the time allotted to try and find a solution; or finally the plans returned may fail if the domain description is too vague and necessary details required to identify failure are missing from the planner's knowledge.

In [de Silva *et al.*, 2009] they incorporate classical planning into a BDI agent architecture, whilst still maintaining the procedural domain knowledge used by the agents, thereby endeavouring to increase the level of the agents autonomy. This is achieved with a combination of pre-specified plans and the ability to produce abstract plans that can be executed using the agents domain knowledge when no applicable plans are available. The planning process starts with a high level of abstraction for general plans before working down to more specialised plans.

2.3.3 Constraint Satisfaction Problem (CSP)

A CSP consist of a set of variables each with a domain of possible values, and a set of constraints on the variables restricting the possible value assignments. As in planning, a consistent assignment is one where no constraints are violated, and a solution to a Constraint Satisfaction Problem (CSP) is a complete assignment where each variable is mentioned [Russell and Norvig, 2003, chapter 5]. Many search algorithms have been developed to take advantage of the structure of CSPs and the common structure allows general purpose heuristics and algorithms to be used on a wide range of problems. A couple of common problems solved using CSPs include the 8-Queens problem and graph-colouring.

Constraint Satisfaction Problem and Distributed Constraint Satisfaction Problem are often used for planning and scheduling or time-tabling, and already have many refined algorithms with proofs of sound and completeness with quality guarantees for the results when the search space makes it infeasible to perform a complete check of all possible solutions. When this is the case, attempting to optimise the value of solutions can be more useful than spending a long time searching for the best solution if they are timely and offer guarantees on the level of optimality. This is offered by Constraint Optimization Problems (COP) and Distributed Constraint Optimization Problem (DCOP) and used by Mailler and Lesser [2004a], Maheswaran *et al.* [2004], Pearce *et al.* [2006], amongst others.

Constraint Optimization Problem (COP) is a variation on CSP when producing a complete solution is intractable. This approach allows a solution to be found without requiring the best solution to be found [Mailler and Lesser, 2004a]. The level of optimality is a trade off between the amount of time available to compute the solution, and the need for a complete optimal solution. A COP consists of a set of n variables, with a domain of finite values for each variable, and a set of cost functions between the values in the domains. The aim is to find a solution where the global cost is minimized. The Distributed Constraint Optimization Problem (DCOP) is the distributed version of this, attempting to find the optimal solution where the variables are spread out among the agents. A soft COP is used in [Thangarajah *et al.*, 2007] to model the whole reasoning process of the agent, rather than that of potential goal effects.

Weigel and Faltings [1999] discuss existing approaches to reducing the computational complexity of solving CSPs through structuring techniques such as Tree Clustering or Hinge decomposition to break the problem down into smaller problems that are easier to solve, before going on to introduce their own hierarchical structuring technique making use of interchangeability and partial solutions. They have also produced algorithms to structure the problems, with examples of where they have used the new approach and some experimental results suggesting the algorithm is effective in reducing the amount of computation required to find solutions.

The job shop scheduling problem, from planning, has also been considered by other approaches including CSPs as is shown in [Cheng and Smith, 1995], showing that CSPs can be used as a major approach to solving scheduling problems [Galipienso and Sanchís, 2001].

CSPs can also be used for solving agent-related problems, such as in the work by Norman *et al.* [2003] where CSPs are used to model the decision making process of agents in a services bidding system, when deciding how much of their service to offer and if they need any additional resources from others, while in [Hannebauer, 2001] they use CSPs to model the internal conflicts within an agent. In that paper they define an architecture for solving the internal conflicts that can also operate in a distributed environment, before using Distributed CSPs to model and reason about external conflicts. The main type of conflict they consider in [Hannebauer, 2001] is the conflict that occurs between two competing goals. These can either be goals within a single agent in the case of internal conflict, or the separate goals of two different agents, externally competing against and in conflict with each other. The main focus of their work is to look at the relationships between internal and external general conflicts and provide an approach through which they can both be solved.

While CSPs have improved the ability to solve many problems they still have some drawbacks relating to complexity and scalability. Alternatives that have arisen from these problems include searching for near optimal solutions rather than exact solutions, or further partitioning the problem and distributing it between multiple automated agents. These approaches also tend to offer more realistic possibilities for use in real-world problems. However, in distributed systems, there

is an extra overhead of communication and coordination that needs to be managed if any benefits are still to be gained from the distribution. A formalisation of Distributed CSPs (DCSPs) is given by Hannebauer [2000]. The paper details an algorithm for partitioning the problem through dynamic reconfigurations so there is a smaller amount of external communication in order to reduce the costs of coordination between the agents.

In [Yokoo and Hirayama, 2000], an overview of various algorithms for solving DCSPs is given, including algorithms such as Distributed Breakout, Asynchronous Weak-Search, and Asynchronous Back Tracking, all as extensions to non-distributed CSP algorithms. It also states that DCSPs are well suited to handling problems of resource allocation between agents, when the resources or tasks are viewed as variables and the possible resource assignments are viewed as values. In the paper, results are given for the various algorithms comparing the number of cycles each took on average to find solutions for various input sizes. The asynchronous backtracking algorithm, which is very static in its distribution of the problem takes considerably more cycles than the other algorithms, and is also unable to solve the larger problems, while the asynchronous weak-search algorithm uses the least cycles for the different problem sizes with sparse interactions between agents. However, in the 'critical' problems with lots of interactions, the asynchronous weak-search performs badly, requiring seven times as many cycles as the Distributed breakout algorithm.

In [Mailler and Lesser, 2004b], an initial description of a new algorithm, Asynchronous Partial Overlay (APO) is given using an alternative approach to solving Distributed Constraint Satisfaction Problem (DCSP) by applying cooperative mediation rather than asynchronous backtracking, which is commonly used in other approaches. This approach is then expanded on in a later paper [Mailler and Lesser, 2006]. In this approach, whenever an agent detects any conflict it forms a mediation session with its direct neighbours. Both papers give the algorithms used to initialize the mediation, perform local resolution and for conducting mediation sessions, with the second paper then giving more detailed examples based on a 3-colour problem and proving the sound and completeness of the algorithm. The algorithm is then compared against the Asynchronous Weak Commitment protocol for solving the distributed 3-colouring DCSP. The performance is measured

based on the number of cycles and messages required to solve the problem, with n variables and m binary constraints. By just looking at the cycles, at low density the two algorithms seem to performed equally, while at higher density the APO algorithm required slightly less cycles, however when you also consider the message count the Asynchronous Weak Commitment (AWC) was very message intensive, particularly on the larger values of n, while APO was significantly more conservative in communication. The third experiment they performed to compare the two algorithms shows the time taken to solve the problems. This shows a considerable difference between the two algorithms, with the APO taking less than 10 seconds to complete the largest problem, while the AWC required 92350 seconds.

Benisch and Sadeh [2005] compare experimentally the tradeoffs between the two main approaches to solving DCSPs, namely Asynchronous Backtracking and Cooperative Mediation. The results use a standardised measure of comparison based on the number of non-concurrent constraint checks (NCCC)and also the number of messages passed. Along with the original algorithms that are being compared, there are also variations on these based on different configurations of the algorithms. The Asynchronous Back Tracking (ABT) algorithms are shown to send considerably more messages than the APO algorithms in higher density problems, while the APO-Branch and Bound (BB) algorithm has considerably higher NCCCs in the higher density problems.

Out of the three possible approaches discussed here, the approaches that are actually developed in this thesis are the application of Petri nets (see chapter 4) and Constraint Satisfaction Problems (see chapter 5), to give two contrasting approaches for evaluation of the reasoning about goals.

2.4 Testing Performance

Once the two approaches for reasoning about goals have been developed, we then need some method of analysing their performance. In this section we look at existing frameworks for analysing the performance of individual agents, to see what methods and results are available from the evaluation of related approaches with which to compare the results given in chapter 6.

The model of a Mars rover gives a real-world testbed that can be used to assess the performance of an agent working on its own, and reasoning about goals. This testbed has been used in related work [Thangarajah *et al.*, 2002, 2003a,b, Clement *et al.*, 2001, Estlin *et al.*, 1999, Matthies *et al.*, 1995, Raja and Lesser, 2004a, Shaw and Bordini, 2008, Shaw *et al.*, 2008] providing a common measure for comparisons between results. The rover can be given two main types of goals, both of which are performed in several locations, these being collecting soil samples and collecting rock samples. The results have to be transmitted back to Earth via a base station, and of course all actions will use different amounts of energy. There may also be a limited amount of storage space for the samples collected that have to be taken back to the base station. All these various factors can be combined together to cause conflicts that need to be carefully controlled and reasoned about in order to avoid them.

The different approaches to the three types of reasoning discussed above each give different sets of performance tests that can be used for comparison. The work by Thangarajah *et al.* gives results for the individual types of reasoning, which show an increase in the number of achieved goals and measure of additional computational costs. In [Raja and Lesser, 2004a], they only focus on reasoning about resources, and measure the utility gained from the reasoning. Clement and Durfee [2000b] present results using the summary information they described in earlier papers. The results cover the reasoning about resources and effects and measures the performance based on the CPU time taken to find an optimal solution. Approaches using CSPs (see section 2.3.3) tend to measure performance based on the number of cycles required to find solutions, while the DCSPs also take into consideration the number of messages passed between agents. As no one else has used CSPs for the reasoning that is performed here, performing this evaluation would not be very meaningful. In addition, this analysis could only be performed on one of the developed approaches, without a related measure from the Petri net approach to compare it against.

As this research builds on the work done by Thangarajah [2004], using a similar approach for analysing the effectiveness of the reasoning will enable the two approaches defined here to be compared to the approach developed by Thangarajah *et al.* Besides a Mars rover, their approach makes use of an abstract scenario

with two different goal structures being used to compare the effectiveness of the reasoning on different goal depths, these being depth 2 and depth 5, and varying different parameters for analysis. The parameters include the amount of parallelism between the goals, as well as the levels of interaction and resource availability. This provides the most suitable form of evaluation when considering the performance of the approaches developed here.

Chapter 3

Reasoning about Goals

In this chapter, we will discuss the goal-plan tree problem and the models for which we are developing reasoning in the following two chapters.

3.1 Goal-Plan Tree

The goal-plan tree structure used and the types of reasoning applied are based on those defined by Thangarajah [2004]. The goal-plan tree consists of a top-level goal at the root, with one or more plans available to achieve that goal. Each of these plans may themselves include further subgoals forming the next level in the tree, followed by additional plans to achieve these subgoals[1]. The simplest plans at the leaves of the tree will just contain a sequence of actions and no further subgoals. An example of a goal-plan tree was shown in figure 2.1, which shows the goal-plan tree representation of a goal for a Mars Rover to collect a soil sample from a location then transmit the results back to Earth via the base station. An agent will most likely have multiple top-level goals to achieve, each with its own goal-plan tree.

When attempting to achieve a goal, the various branches in the tree can be thought of as either AND or OR branches. Where a goal or subgoal has a range of applicable plans, only one of these plans needs to be completed for the goal

[1]The term subgoals will always be used when referring to subgoals, while top-level goals will either be referred to as goals or top-level goals

or subgoal to be achieved, as in an OR branch. However, where a plan has one or more subgoals, then all of these subgoals must be achieved for the plan to be successful, so these can be considered as an AND branch. This hierarchy gives rise to the goal-plan tree structure that we can then use to represent goals in order to reason about them.

While the goals simply have an ID and list of possibly relevant plans, each of the plans may have a set of preconditions that must be true in order for them to be applicable at a given time and a set of effects that they are likely to achieve within the environment. These can be represented as two possibly empty sets, one for preconditions and one for effects. The plans, or more precisely the actions within the plans, may consume some resources in order for them to be executed. For the purposes of this thesis and for the reasoning used here, it is assumed that the actions are successful at achieving the desired effects, however as this is not necessarily a realistic assumption for agents to make, in future work this could be extended to take into consideration plan failures. It is also assumed that the information regarding the amounts and types of resources required can be accessed easily, for example a summary kept by the plan along with the lists of preconditions and effects.

Within an individual agent there may be one or more top-level goals for the agent to achieve. While it is often straightforward for these to be achieved in sequence, it may be possible for the agent to achieve better performance by attempting to achieve them in parallel. This can of course lead to problems where the goals interfere with each other and where resources are limited so reasoning needs to be applied for the agent to be successful. The three types of reasoning considered in this thesis are based on: 1. the limited availability of consumable resources, 2. the potential for positive interactions between goals and 3. the risk of negative interference between goals, which are discussed in more detail in the following sections.

3.2 Consumable Resources

There are many things that can be considered as resources and these can be split into various sub categories such as reusable or consumable resources. An example

of a reusable resource would be a communication channel, while an example of a consumable resource would be energy, money or even time. While it is possible to "recharge" resources such as energy through a solar panel for the Mars Rover example given above, it is assumed in this thesis that once a consumable resource has been "consumed" then it is no longer available and will not return at any point in the future. Future work could extend the reasoning to include maintenance goals, where a desired state in the environment is maintained for a length of time, that would take the recharging into consideration. However, the focus for reasoning about resources is to endeavour to make the best possible use of resources when there is a limited supply.

In terms of reasoning about resources, this thesis focuses only on consumable resources as it is expected that introducing reasoning about this form will provide the greatest improvement in overall performance when measuring performance based on the number of goals achieved. This is because the use of reusable resources can be scheduled, taking into account priorities between goals if necessary, however the use of consumable resources is constrained by the quantity available. Once it has been used it cannot be reused, so it is important to avoid wasting any by adopting goals that cannot be achieved with the available resources or by poor choice of plans. The choice of plans is important when there is more than one plan available to achieve a goal or subgoal. If one of the plans consumes a large amount of resources, while the other consumes very little, it will be preferable to select the plan with the lower resource requirements. In addition, when considering consumable resources such as fossil fuels, it is becoming increasingly important that we attempt to make the most effective use of the resources we have available.

For example, if an agent had two goals where one goal required 70 units of resource and the other 60 units, it would not be rational for the agent to start both goals if there were only 100 units of resource available. If a third goal was added that only required 40 units, then it would be feasible and may be preferable to achieve the two smaller goals, rather than the single larger goal. Equally, if there were two plans, both capable of achieving the same results, one of which required 10 resources while the other only required 5, then provided there are no other restrictions, it would be rational for the agent to select the plan with the lower resource requirements to conserve resources for use elsewhere.

The resources involved are consumed by the individual plans, or more precisely, the actions within the plans. Therefore, when a goal and its plans are defined, the total resource requirements for that goal are unknown. As not all plans will be needed due to the plan choices available at subgoals, calculating the total resource requirements is not as straightforward as simply summing up the resource requirements for all the plans in the given goal. The result of the different choices for plans selected at subgoals means that *best* and *worst case* resource requirements can be calculated. The best case is that involving the lowest resource requirements of all possible choices, with the possibility of considering weightings of different resource types where necessary, while the worst case is the branch consuming the greatest amount of resources. If one type of resource is particularly precious, then it may be preferable to use large amounts of a cheaper or more abundant resource to preserve the precious one. This could be indicated by applying a heavy weighting or cost to the precious resource and a very low weighting to the cheaper type.

While in the positive and negative types of reasoning we are able to avoid the use of summary information used by Thangarajah *et al.*, it is not feasible to completely avoid this here. The amount of summary information has been significantly reduced when compared to that used in [Thangarajah *et al.*, 2002], to either a single number, or simply a list containing the amount of each type of resource used. The type of summary information used depends on the point at which the information is being used and is described in more detail in section 4.3

Calculating the resource requirements for a given goal can be achieved by starting at the leaves of the tree. These resource requirements can be propagated up to the root, giving a best and worst case resource requirement for each goal. Depending on whether the branch reached is at a plan or at a subgoal affects how the requirements are added up. At a plan branch, the requirements from each of the leaves are simply added together, along with any resource requirements from the plan itself, to form the new total that can be propagated up, while at a subgoal branch the lowest requirement is set as the best case and the highest as the worst case to propagate up. Once at the root, these values can then be applied to the goal selection to ensure goals are only started where there are sufficient resources available to complete them. The generated summary information can also be used for the plan selection when there is a choice between multiple applicable plans for

achieving a goal or subgoal, opting for the best case wherever possible.

Where there is more than one type of consumable resource involved, the resource reasoning can be done in one of two ways: firstly by just summarising the individual types of resources to produce a list of the requirements for each resource, or secondly by generating an overall summary value for the different types of resources. In this second case in particular, it may be desirable to apply weightings to the different types of resources to indicate their respective costs.

When evaluating the performance of this type of reasoning, it is expected that the largest impact on the amount of resources used will be where there is a large amount of branching within a tree. This will allow the greatest application of the best case branch selection within the goal-plan tree. Varying the number of top-level goals, where each goal has slightly different resource requirements is also likely to provide a method for stress testing this approach, so will form part of the focus for the evaluation of this reasoning (see chapter 6).

In the work by Thangarajah *et al.* [2002] to reason about resources, they use additional summary information to identify possible conflicts between goals based on their requirements. The summary information generated includes normalised lists, containing each type of resource precisely once, detailing how much of each resource is required by a given goal. The resources considered include both consumable and reusable resources used by goals.

These lists are then split into "necessary" and "possible" lists, the "necessary" list containing the resources that would definitely be required, with the "possible" list containing the list of resources that may or may not be used. A set of operations are defined for combining the different resources in order to identify the necessary and possibly necessary resource requirements. This is applied in places where there is branching in the tree with a choice of applicable plans to aid the selection of plans in an attempt to make the best use of the available resources. One can start forming these lists from the leaves of the tree, and by passing up the details towards the root. At each level the operations are applied to combine the requirements of the plans at the current level with those of the sub-plans, until the root of the tree is reached and the overall resource requirements for each goal

are known. During the execution of the reasoning, resource information is used to identify resource conflicts for example in the use of reusable resources that may need to be sequenced if there are insufficient resources available for them to execute in parallel, or goals that cannot safely be started due to insufficient consumable resources available. The reasoning is also used to decide whether it is safe to accept a new goal, avoiding new goals that will conflict with existing consumable resource requirements. If goals are only possibly conflicting based on the possible requirements of the goals, then bold agents may choose to start the new goal and simply monitor the goals for signs of actual conflict.

Results from the application of this reasoning can be seen in [Thangarajah and Padgham, 2004], with a comparison between their results and those presented here given in chapter 7.

While the approaches developed here are based on some of the ideas presented by the work of Thangarajah *et al.*, there are some key differences, which are discussed over the next two chapters where the two approaches are discussed in more detail.

3.3 Positive Interaction

When two or more goals are being achieved simultaneously by an agent it is possible that there is some overlap or interaction between the plans and the effects caused by the plans in each of the two goals. This can be either beneficial to both goals or detrimental, possibly causing one or both goals to fail.

Positive interaction occurs when two or more plans for different goals achieve the same effects, possibly also using the same preconditions. By identifying these plans, it is possible to select just one of them to execute and to drop the subgoals and sub-plans of the plan(s) not selected. When this interaction occurs high up, near to the root of the tree, this can have a large impact on the number of plans required to achieve a set of goals as the sub-tree of the removed plan or plans can itself potentially contain many plans.

An example of this occurring within the Mars rover example is when multiple goals have the objective of obtaining samples from the same location and transmitting the results back to the base. If the goals were executed in sequence then

the rover would move to the location, get the first sample then return to the base to transmit the results before starting the second goal which involved returning to the same location to obtain the second sample. By taking both samples at the same time the rover can save a lot of time and energy and transmit both sets of results back at the same time.

The interacting plans can be identified firstly by considering the effects of the different plans. Where two plans achieve the same effect then only one of these plans should need to be executed for both to be achieved. It may be possible that the two plans have different preconditions as they are designed to work in different situations, while still achieving the same results. The preconditions of the remaining plan still need to be achieved before the plan can execute to achieve the effects for the interacting goals.

In the evaluation of this type of reasoning, the level within the goal-plan tree at which the interaction occurs will be an important factor in assessing the performance of this reasoning. High-level positive interaction occurs between plans that are near the root in the tree interact and achieve the same effect, while low-level positive interaction occurs near the leaves in the trees. It is the high-level positive interaction that is expected to have the greatest impact on the number of plans used, as the plans dropped will contain the greatest number of subgoals and plans. This number of plans saved in the sub-tree that is dropped will also depend on the degree of branching at the subgoals and plans, with greater savings available when subgoal branching is low and plan branching is high. This is because all the subgoals of a plan need to be achieved, while only one of the plans for a subgoal needs to be achieved for a goal to be successful.

When combining this reasoning with reasoning about resources, the resource requirements of the different plans can be taken into consideration to select the plan with the lower resource requirements. It is anticipated that when these two types of reasoning are combined, the effects will be particularly significant as the amount of resources consumed will be reduced proportionally to the number of plans being used. The anticipation is that this will then allow further goals to be started that would otherwise not have had sufficient resources available to be safely adopted.

The positive interaction reasoning performed by Thangarajah *et al.* is based on the effect summaries of plans, maintaining lists of "definite effects" and "potential effects" [Thangarajah *et al.*, 2003b], with similar meanings to those of "necessary" and "possible" resources described in the previous section. The "definite effects" are those that will be achieved at some point in every possible path option available for achieving a given goal, while the "potential effects" refer to those effects which just appear in at least one, but not all, possible paths through the tree. These are based on the summary information used by Clement and Durfee [1999a] to reason about negative interference between goals.

As with the resource reasoning, the summary information, in the approach by Thangarajah, is generated by propagating information about plan effects up from the leaves in the tree to the root, recording the plans that bring about each effect. For a definite effect, this could potentially be a large number of plans if there is a lot of branching within the tree to choose between. However, if just one of these plan options for a branch is missing the effect, then the effect drops down to a potential effect, as there is at least one selection of plans within the tree that satisfies the top level goal without achieving this effect.

Once these lists of summary information have been generated, they can then be used to identify plans that can potentially be merged. This takes into consideration whether a plan will definitely be needed by a goal, or whether it is an option at a branch, to define further lists of "Definitely Mergeable Plans" (DMP) and "Possibly Mergeable Plans" (PMP). A "Waiting Goals List" (WGL) is then also needed to prevent deadlock from occurring, as goals are suspended while plans are waiting to be merged. This deadlock could occur if two interacting goals are suspended waiting for each other to reach another merging point. If this happens, the attempt at merging plans at this point will be dropped and the plans executed as normal. In order for plans to be listed in the DMP, they must include effects that will definitely be achieved by all the goals interacting based on these effects. If the effect is only a potential effect in one of the goals, then they are placed into the PMP list.

The timings of the merged plans are important for this reasoning. When a "mergable" plan is reached in the plan schedule, the plan is flagged as ready to execute and the goal is suspended until its counterparts in other goals are ready.

However, if this is likely to take a long time, particularly where one of the effects involved is only a potential effect, then the attempt to merge the plans will be dropped and both plans will be allowed to execute as they would normally have done.

When two plans achieving the same effect are finally ready at the same time, then the agent has to decide how to "merge", if still possible, the plans. This is essentially choosing between the two plans to decide which one to execute and which to drop. In most cases, either plan can be selected, the problems arising if one or both of the plans achieve additional definite effects which the other plan does not. As the effect definitely needs to be achieved for the goal to be successful, despite waiting for the other plan to be ready, it may still be unfeasible to merge the plans at all if both plans have definite effects that are not achieved by the other.

After every merge, the lists identifying mergable plans are updated to remove the plans that were "merged". An update check is then performed looking for any additional interactions in the remaining plans related to the effects that caused the merging.

While the results for the positive interaction reasoning presented in [Thangarajah, 2004] show that it is effective, the sizes of the goal-plan trees used were kept quite low. The large number of lists, and the level of detail stored in each could potentially grow exponentially as the tree sizes, number of goals and thus number of effects involved increases, resulting in prohibitively large reasoning overheads. This is one of the main reasons for looking at developing approaches that avoid the use of such large quantities of summary information.

In the case of the reasoning performed here in this thesis, the focus is on the effects being achieved for each goal without synchronising the execution of the interacting plans between goals. Once an effect has been achieved by a plan, it does not need to be achieved again by another plan unless the effect is lost, such as when the Mars rover leaves the location it is currently at to move to another or back to the base station. Therefore, the timing is not necessarily as important, provided the achievement of the effect does not negatively interfere with any other goals and plans that could cause it to cease to exist after the interference, as discussed in the next section.

3.4 Negative Interference

Opposite to the positive interaction, the final type of reasoning that is covered by this thesis is regarding negative interference between three or more plans, of which at least two will be within the same goal. This occurs when there is a causal link between two plans for a given goal, where the effects of the first plan sets up the necessary preconditions for the second. If a third plan, usually from a different goal, with opposing effects to the first plan, attempts to execute between the two causally linked plans, this can result in the link being broken and lead the pair of plans to fail, potentially resulting in the failure of the goal which they were attempting to achieve and wasting any resources that had already been consumed by the goal. By avoiding negative interference, it is possible to prevent the interference from happening and in doing so achieve more goals that would otherwise have failed as a result of the interference.

For example, using the Mars rover example again, negative interference can occur when two or more goals require taking samples at different locations. If after having moved to the first location, a second goal interferes to take the rover to another location before the sample is taken to satisfy the first goal then the first goal would fail unless the step of returning to the first location was repeated, wasting both energy and time. To avoid this, the causal link between the plan generating the effect and the plan making use of the effect needs to be identified based on the effects and preconditions of the two plans. Other plans that then try to change the effects involved before the second plan has been completed need to wait until it is safe again for them to execute.

There are two ways in which the interfering plans can be safely scheduled. This is either by executing them before the causal link or after. Which is used will also depend on whether any plans rely on the effects generated by the interfering plan. In the worst case, the goals could be required to execute in sequence as no safe parallel scheduling can be found without some plans interfering with each other.

As with the positive interaction, it is the level, or in this case the distance between the causally linked plans that is expected to have the greatest impact in the number of goals achieved without any reasoning, thereby putting the reasoning under the highest level of strain with high levels of interactions between the goals

as well. This distance can be thought of as a duration, by which it is meant as the length of time between the plan setting up the effect and the effect actually being used closing the link, rather than any execution time taken by the plans themselves. A deep tree where the duration is likely to be longest, with the greatest distance between effects achieved near the root of the tree and those effects being used at the leaves, is therefore the main place where this reasoning is going to be tested. This will be done by varying the amount of interaction between goals and the duration for which the effects need to be protected by adjusting the depth in the tree where the effects are achieved and then used.

This reasoning is again based on that defined by Thangarajah *et al.* [2003a]. In addition to the preconditions of plans mentioned above, they also include *in-conditions* as a subset of preconditions that must be maintained for the whole duration of the plan executing for the plan not to fail. While the in-conditions are not explicitly stated in the approaches developed here, both models maintain the necessary conditions while a plan is executing.

While they consider the causal links between plans within a goal, they also include some reasoning to ensure the in-conditions are also protected for the whole duration of the plan executing. In order to do this, they not only need to maintain the effects summary lists as defined for the positive interaction reasoning, but also a set of lists for pre- and in-condtions of plans. Only the preconditions brought about by the effects of other plans need to be protected by this reasoning to prevent interference between goals.

The terms “definite” and “potential” are applied to the conditions in the same was as they were previously used for effects, and now refer to both the effects and pre-conditions produced and required by plans. So, for example, a definite condition will definitely be produced by a plan and definitely be required as a precondition of a later plan. There are now six separate lists for definite and potential effects, in-conditions, and preparatory effects for pre-conditions. To protect any necessary conditions whilst goals are executing in parallel a “guarded set” is introduced containing the in-conditions and dependency links that are currently active. These are protected from other plans and goals, which are forced to wait until the conditions or links they would interfere with have been removed from the guarded set.

While in this previous work by Thangarjah *et al.* the use of summary information was applied to perform the various types of reasoning, the approaches developed in this thesis aim to remove the need for much of this information, just maintaining a minimal amount required for the reasoning about resources. The aim is to reduce the overhead of storing potentially exponential lists [Clement and Durfee, 2000a], which could cause difficulties when reasoning about large numbers of large-sized trees, especially for the large number of lists that are maintained for the approach to negative reasoning by Thangarajah *et al.*

Combined Reasoning In previous work where different types of reasoning have been considered, they have generally just been analysed on their own to test their effectiveness on performance without any reasoning. While this is also being done here, the aim is to also consider the combined effectiveness of the three types of reasoning together to analyse how the different types of reasoning interact with each other, both in pairs and with all three types combined together.

Given the amount of summary information stored individually for the positive and negative types of reasoning in the work by Thangarajah *et al.*, combining these two reasoning types would be likely to cause large overheads in terms of storage, processing and monitoring of the array of lists needed by the reasoning.

In [Shaw *et al.*, 2008], a brief summary is given for a proof showing that the problem of reasoning about the goal-plan trees, using the tree structure and a "weaker" abstract version of the reasoning types described above, is NP-Complete. While this means finding an exact solution to instances of the problem could be very costly in terms of time taken, we later experimentally show that it is feasible to perform reasoning that is effective in a reasonable length of time.

3.5 Goal-Plan Tree Automated Generation

In order to perform a reasonable set of experiments, using a variety of goal-plan trees and with multiple different settings, it was necessary to allow for automated generation and processing of the goal-plan trees, and to produce a standardised

format that could be provided as an input to the Petri net and constraint-based models to produce the different instances of each model for a given tree structure. As a result, the goal-plan tree has been represented in an XML format, the definition of which is shown below.

The Data Type Definition (DTD) defined for representing the goal-plan tree in XML format is as follows:

```
<!ELEMENT goal (plan)+>
<!ELEMENT plan (subgoal*,precondition*,effect*,resource*)>
<!ELEMENT subgoal (plan)+>
<!ELEMENT precondition (#PCDATA)>
<!ELEMENT effect (#PCDATA)>
<!ELEMENT resource (#PCDATA)>
```

This states that each goal and subgoal element must have at least one plan, while a plan can contain zero or many subgoals and sets for preconditions, effects and resources. The preconditions, effects and resources are represented as plain text; however, these are parsed by the Petri net and constraint models to generate the necessary representations.

Template representations of the various tree structures to be used in the experiments were defined in XML that could then be used to generate a large number of top-level goals, inserting variations in effects and resources where necessary. While at this time the XML template representations of the goal-plan tree are generated manually, it is planned in future work to be able to export the plan libraries and top level goals from a selection of agent programming languages to generate the XML that can then be passed through to the different reasoners. This intermediate XML level means that the reasoners are not tied to specific languages, so provided an export function could be defined that generates this predefined XML structure then any goal-plan based agent language with programmer-defined plans could potentially make use of the reasoning. It is also possible that agent languages that generate plans could potentially apply this reasoning with further work to allow the dynamic addition of plans as well as goals.

In chapter 6, the two reasoning approaches developed are compared, and evaluated to consider situations where one approach could potentially provide better

results over the other. Once these situations affecting performance are identified, the information could then be used to suggest which reasoner would be more applicable to a given application based on factors such as tree structure and levels of interaction amongst others.

To test the performance of the two approaches, three different tree structures, and variations on the tree sizes of each were used in the experiments. As a result, it was possible to define XML templates to represent the goal-plan tree structure of a single goal that could then be replicated the desired number of times to generate sufficient goals, with variations in preconditions, effects and resource requirements being applied to each.

Once the XML goal-plan tree representation has been defined for the required number of goals, this is then parsed using an XML SAX parser, which uses the data to generate a list of goal-plan tree objects where each goal, subgoal or plan is represented as a node with a, possibly empty, list of children and variables to store preconditions, effects and resources, along with an ID for each node. The automated parsing of the XML representation, and the generation of each of the reasoning models was achieved using Java 1.5, with the Eclipse 3.4 IDE.

The top-level list of goals is passed on to the models for generating each of the reasoning models, which traverse the goal-plan trees defined in the list. This is discussed in more detail in chapters 4 and 5.

Chapter 4

Petri net Model

In this chapter we present the first of the models developed here to represent and reason about the goal-plan trees as described in chapter 3. An overview of Petri nets is given in section 4.1 and it is the diagrammatic representations of the flow of control in Petri nets that provides a natural mapping onto the goal-plan tree problem as shown in section 4.2. Sections 4.3 to 4.5 describe how the reasoning can be modelled using Petri nets and how this is incorporated into the Petri net model for goal-plan trees. Finally section 4.6 describes how the generation of the Petri nets for this model can be automated.

4.1 Petri nets

Petri nets are mathematical models, with an intuitive diagrammatic representation, used for describing and studying concurrent systems [Peterson, 1981, Murata, 1989]. They consist of *places* that are connected by *directed arcs* to *transitions*, with *tokens* that are passed from place to place through transitions. The arcs can be inscribed with weightings indicating the number of tokens transferred at a time along that arc. Transitions can only *fire* when there are sufficient tokens, indicated by the weightings, in each of the input places, thereby acting as pre-conditions for the transition. Tokens are then removed from each input place, and one, or the number indicated by the weighting on the outward arcs, is placed in each of the output places. Places are graphically represented as circles, while transitions are

represented as rectangles (see figure 4.1). The tokens are commonly represented as dots on the places, often showing the initial marking or a step in a simulation. These simulations show the flow of tokens through a net and can be used for considering reachability of states or target places within a net.

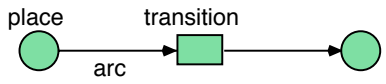


Figure 4.1: Example of a simple Petri net

There are many variations on the basic Petri net representation, and many of these have been applied to a variety of agent systems [Mazouzi *et al.*, 2002, Bonnet-Torrès and Tessier, 2005]. A common variation is the addition of weights to arcs as described above, with the default weight being one. Greater weights on arcs either require the place to have at least that many tokens for the transition to fire, removing the specified number from the input place, or the transition adds to the output place that number of tokens as its output. A selection of different arcs are also introduced in addition to the single directed arcs; these include negated arcs and bidirectional arcs [Christensen and Hansen, 1993]. Negated arcs should only be used from places to transitions, indicating that the place should not contain any tokens in order for the transition to fire. The bidirectional arc between places and transitions results in the place being both an input and output place for a given transition.

Another variation is that of coloured Petri nets which are able to hold tokens of different types, representing for example different data types [Kristensen *et al.*, 1998]. The inscriptions of weightings on the arcs can then be used to identify the type of tokens required, as well as the quantity of each to ensure the appropriate tokens are transferred when the transitions fire. The transitions can also be inscribed with conditions indicating when it can fire, ranging from mathematical functions to analysis of the contents of the tokens where tokens could contain a list of values.

Finally, Reference nets allow Petri nets to contain instances of sub-nets that are passed around in the same way as other tokens in a net. When a reference is made to another net this is done by a pair of transitions with matching reference

inscriptions, one in each net, that act to synchronise the two nets. The inscriptions can contain parameters that allow variables and tokens to be transferred between the two nets. By synchronisation, it is meant that the transition in the referencing net acts as an additional trigger for the transition in the referenced net to fire. Both transitions need all preconditions in the form of tokens in input places in order for the pair to fire. Once the pair has fired, the two nets then continue to fire any available transitions in sequence until the next synchronisation point.

Renew is a Petri net editor and simulator that is able to support high-level Petri nets such as coloured and Reference nets [Kummer *et al.*, 2006], and is the editor of choice for modelling the Petri net approach to reasoning about the goal-plan tree problem developed here.

Renew Petri net Notation The basic data types used in Renew as tokens are integers and strings of text, where multiple tokens in a place can be separated by semicolons. Ordinary tokens are represented using `[]` (i.e., empty lists), and strings of text are differentiated from variable names by enclosing the string in double quotes. Multiple values can be stored in a token through the use of a list, which is represented as a series of comma separated values between square brackets. These lists can themselves contain sub-lists as elements, for example `[["A",5],["B",8],["C",21]]`.

When moving lists between places and transitions, it is possible to move the list as a single variable or to open the list to gain access to some or all the elements in the list. Considering the list given as an example above as the token, an arc inscribed with a single variable, e.g. `v`, would generate the assignment of the whole list to `v` when moving the token, while an inscription of `[a,[b,c],d]` would assign the variables to `a=["A",5], b="B", c=8 d=["C",21]`. Each of these variables can then be passed on as separate tokens to the output places, or merged back into a single list.

Transitions can be inscribed with operations to perform on the variables, or conditions that must be met by the values assigned to the variables in order for the transition to fire. Simple operations could include taking two numbers as input tokens from two places and adding them together with the total being placed in one or more of the output places. Conditions can also be inscribed on the transitions,

such as checking the sum is greater than a value. However, in order to ensure the condition is met, the keyword *guard* is required before the condition. There are two main types of expressions that can be used in the inscriptions on transitions, these are conditions and assignments. An example of this is shown in figure 4.2 where each of the two input places has two integer tokens and the transition has the inscription to sum two numbers, with a `guard` condition ensuring the total is greater than 10. When a simulation of this net is run, the tokens 5 and 7 are selected from the input places to satisfy the condition.

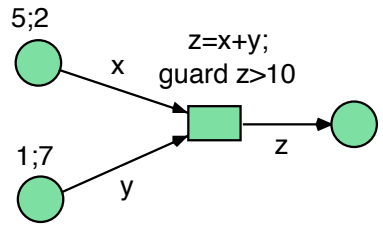


Figure 4.2: Petri Net example of inscribing transitions with operations and conditions

Reference nets, as described above, allow Petri nets that are potentially in different files to refer to each other and to pass variables between them. In the Petri net model developed here, three files are used to hold different components of the model. These are described in more detail in section 4.2.

When creating references to other Petri nets it is possible to generate multiple instances of the same Petri net, each of which is passed around as a token. These tokens referring to other nets can also be passed between multiple nets in the same way as any other tokens. To create a new instance of a Petri net that is in another file the notation used consists of `variableName:new filename` or `variableName:new filename(parameters)` where parameters are being provided.

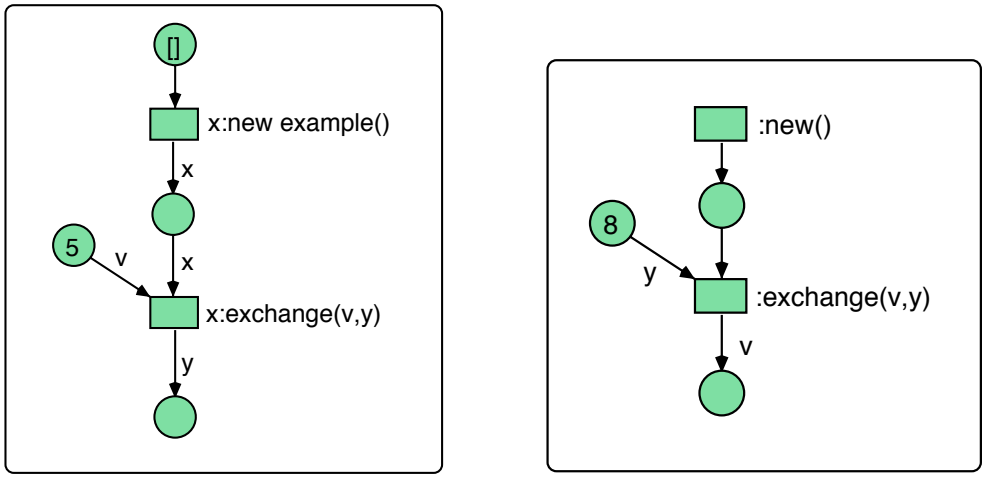
Within the file of which an instance is being created, if parameters are being passed then a transition inscribed with `:new(parameters)` is required in the Petri net, and this transition must be able to fire, (i.e. have sufficient input tokens), when the first net is attempting to create an instance of it.

Once instances of the referenced nets have been created, the referencing of

them is handled through the use of the token containing the reference to the other net and the inscription of a transition in the referenced net. For example, the referencing net would contain the inscription `x:sync(v)` on a transition while the referenced net would contain the inscription `:sync(v)` on a transition. The effect of this is to fire both transitions in the two nets at the same time, thereby synchronising them. In order for this to be successful, both transitions need to have sufficient input tokens to be able to fire. An illustration of this is given in figure 4.3 where values are passed between the two nets. In this example, the simulation starts in figure 4.3(a) where the first transition refers to the top transition in the example.rnw Petri net file shown in figure 4.3(b). This generates a new token `x` in figure 4.3(a) containing a reference to an instance of the Petri net in figure 4.3(b). Multiple different instances of the same Petri net could potentially be created by repeatedly firing the first pair of transitions if the net in figure 4.3(a) had sufficient input tokens. Simultaneously, the top transition in figure 4.3(b) fires, placing a token in its output place. This then enables the next pair of transitions linking the two Petri nets together. If initialisation parameters needed to be passed between the two Petri nets, these could be included in the brackets provided the number of variables in both parameter lists matched, as shown in the second pair of transitions. In a more complicated example, different numbers of places and transitions could be fired between the two synchronising transitions. In the second pair of transitions, each Petri net has a single variable from the parameter list. When the transitions fire, the values are synchronised between the two nets so both nets have both values. However, in this example, only the value from the other Petri net is passed onto the output place, losing the original value that was removed from the left-hand places when the transitions were fired.

4.2 Modelling a Goal-Plan Tree Problem

We have developed a method to represent an agent's goals and plans using Petri nets. Essentially, we are able to represent the same problems as expressed by goal-plan trees in the work by Thangarajah *et al.* (see figure 4.4 for an example of a Petri net representation of the Mars Rover goal-plan tree shown in figure 2.1). In the method we have devised, goals with their branching plan options the sequences



(a) Referencing net (b) Referenced net: example.rnw

Figure 4.3: Petri net example of reference net synchronisation

of actions forming plans and their subgoals are represented by a sequential series of places and transitions. A plan consists of a sequence of actions each of which starts with a place, and has a transition to another place. These transitions represent each of the atomic actions that occur in sequence within that plan. Goals are also set up as places with transitions linked to the available plans for each goal or subgoal. In Figure 4.4, the plans are enclosed in dark boxes, while the goals and subgoals are in light boxes. The plans and subgoals are nested within each other, matching the hierarchical tree structure of the goal-plan tree.

The tree structure is defined with places and transitions such that where there is a plan branching with multiple subgoals then all of these must be achieved. However, when there is a subgoal with a selection of plans to choose from, just one of the plans can be selected. This is shown in figures 4.5(a) and 4.5(b). In the case of the subgoal branch, the top place only ever holds one token. When one of the transitions for a plan option fires, this token is removed thereby preventing the other branch from firing, while in the case of the plan branch, the transition from the plan branch places a token in each of the subgoals allowing all of the branches to proceed. In this model, provided the preconditions for the plans to achieve each subgoal have been satisfied, then the subgoals can be executed in any order. At the bottom of the tree, these branches can then be tied back together using a

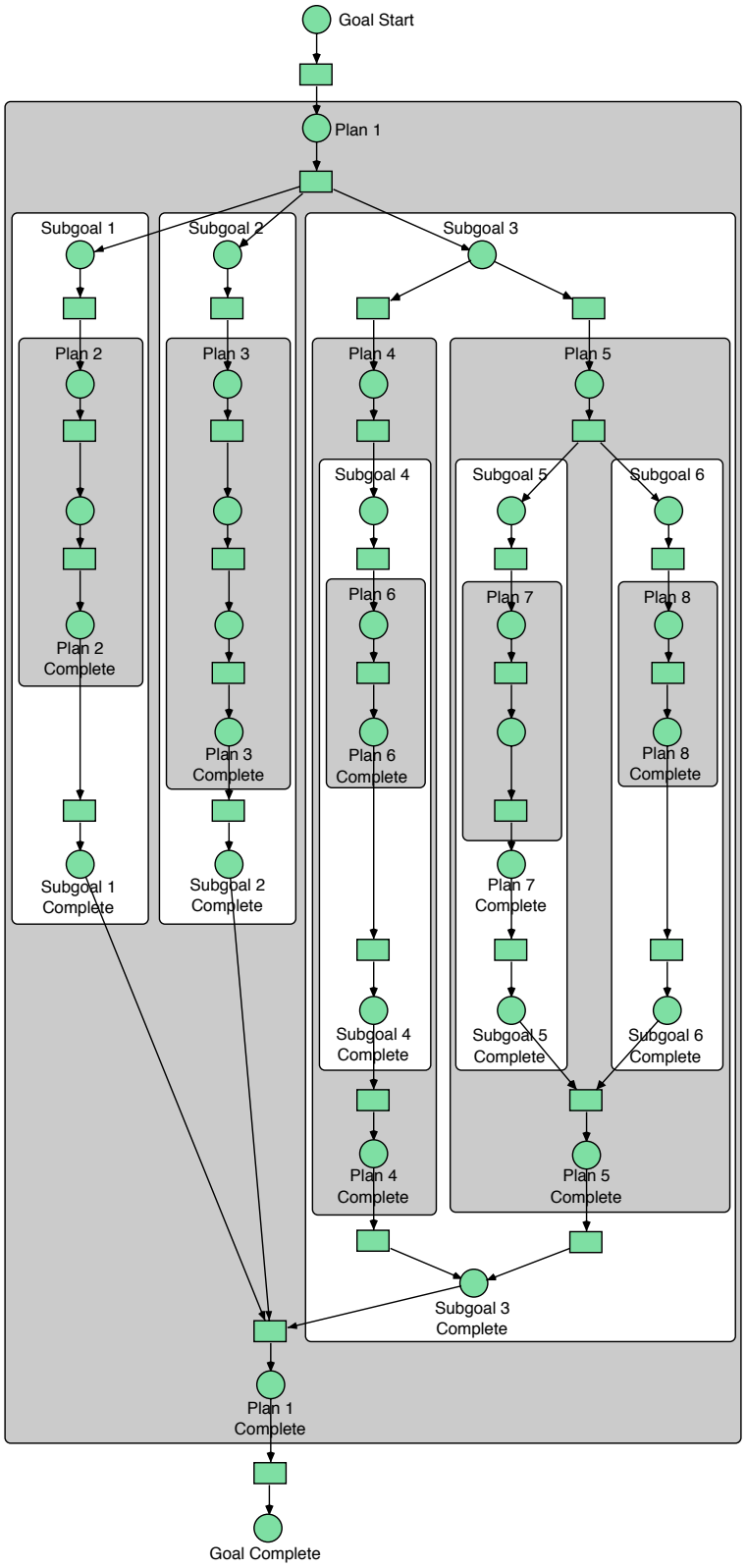
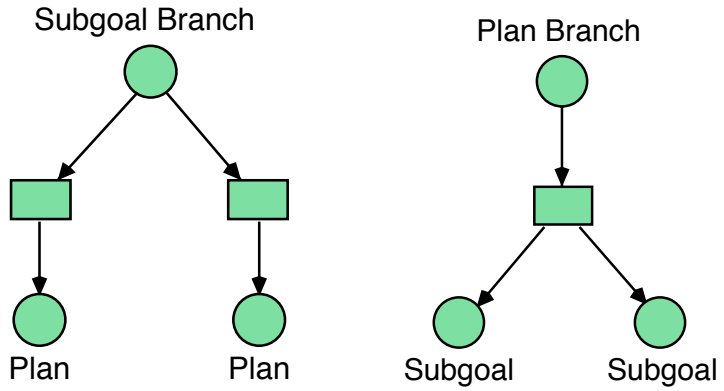


Figure 4.4: Petri Net Representation of the Mars Rover goal-plan tree shown in figure 2.1

similar method in order to ensure all the plans and effects of these plans, or the effects required by the goal, have been achieved. Where there has been a subgoal branch, there will again be separate transitions for each of the plan branches to tie them back together. Once the chosen path has been completed, the transition at the end of that path will fire to deposit a token in a "completed" place to mark the subgoal as having been achieved. Conversely, for the plan branch, there is just one transition at the end of all the subgoal branches, requiring each of the branches to have been completed before the plan itself can be marked as completed as can be seen in the lower part of figure 4.4.



(a) Subgoal Branch. (b) Plan Branch.

Figure 4.5: Petri net representation of the two branch structures used in a goal-plan tree

A simulation of a top-level goal that has been adopted by an agent starts with a token being placed in the "Goal Start" place and then follows the paths taken by the tokens through the different plans to see if the "Goal Complete" place is reached. If it is, then the goal can be considered to have been successfully achieved. However if something prevents transitions from firing along the way, this can be viewed as a plan failing that could ultimately result in the goal failing to be achieved as well. This failure is most likely to be caused either by negative interference from other goals, or by running out of resources.

The Petri net model is split across three files. The first of these contains the

representations of the goals themselves with the reasoning as described in sections 4.3 to 4.5. The factors from the environment that the agent interacts with are represented as a set of variables and are stored in the second Petri net file. The final file is a manager file that oversees the adoption of goals and recording of the outcomes. Within the goals Petri net, a token providing a link to the environment variables net is stored in a place named "Variables" with transitions able to reference this net when required. These variables are represented as places that store values representing the current state of that attribute within the environment, for example, a variable identifying the current location of the Mars Rover could contain an identifier for the current position of the rover.

The goal reasoning that we have incorporated into the Petri nets allows an agent to handle both positive and negative interactions between multiple goals, as well as reasoning about the limited availability of consumable resources. While in the work by Thangarajah *et al.* and Clement *et al.* "summary information" was used in the process of reasoning about goals, it has been possible to avoid this in both the positive and negative reasoning models used within the Petri net. However, when reasoning about consumable resources, some summary information is still required. A comparison of the summary information used is given in section 4.3.

Each of the types of reasoning and the representation of the plans and goals themselves can be viewed as inter-linked modules, as will be discussed below. This modularisation of the method used to represent goals and plans as Petri nets potentially allows an agent to dynamically produce the Petri net representations of its goals and plans, that could be linked into any existing goals and their plans. These representations can then be used by the agent to reason on-the-fly about its ability to adopt a new goal based on its current commitments towards existing goals. This approach also allows the types of reasoning to be selected so the agent need not include all types. It also keeps the reasoning types independent so they do not cause any unexpected interactions with each other.

The Petri nets generated by an agent, as previously stated, can be used to advise on the adoption of new goals, and also the plan selection for both existing goals and any new goals that are accepted. The plan selection process aims to avoid any potential negative interference while making use of plans that can benefit from positive interactions and the selection of plans with lower resource requirements

to make the best use of available resources.

4.3 Modelling Consumable Resource Reasoning

As stated above, each of the types of reasoning can be represented as a separate module that can be "plugged-in" to the basic Petri net representation of a goal-plan tree at the relevant locations. These locations span both the plans that are generating effects and also a layer of interaction around the environment variable being affected.

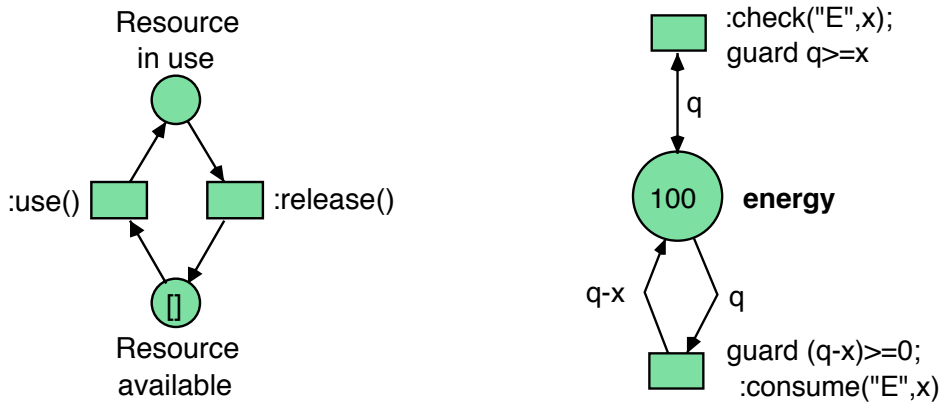
The interactions between the goals are modelled using a set of *variables* that make use of the properties of coloured Petri nets to contain numbers or letters, used for representing different states of the environment. Each variable represents a different property whose value can be changed during the lifecycle of the environments simulation.

While in sections 4.4 and 4.5 it will be shown that it is possible to avoid the use of summary information when reasoning about positive and negative interactions using a Petri net approach to the goal-plan tree problem, this is not possible when reasoning about resources. However, we show here how we have minimised the use of summary information compared to the levels used in [Thangarajah *et al.*, 2002]. We only use a compact form of summary information where it is absolutely required and this allows us to gain a significant improvement on the resource usage when compared to no reasoning being employed.

Summary information is used in two ways. Firstly, a summary of all the resource requirements is produced and used to decide if a goal can be safely taken on, based on existing resource availability, and secondly, where a goal or subgoal has a choice of plans, summary information just for the subtrees is provided so as to select a preferred plan (i.e. the one with the lowest resource requirements).

As stated in the previous chapter, there are two main groups of resources that could be considered when reasoning about resources. These are reusable resources and consumable resources. An instance of a reusable resource can only be used by one plan at a given time, but when that plan has finished executing, the resource is available again for another plan to use it. A typical example of such a type of resource is a communication channel. On the other hand, consumable

resources can only be used once, and then no longer exist, for example units of energy or time. Reusable resources can be represented as shown in Figure 4.6(a). In this Petri net, a single token representing the resource is passed between two states representing its availability. However, the reasoning presented in this thesis refers solely to reasoning about *consumable* resources. Figure 4.6(b) shows the basic representation of consumable resources within a Petri net. The central place contains a numeric token indicating the initial quantity of the resource available. Two transitions are then provided for referring to the resource: firstly checking how much is available without consuming any and secondly consuming some of the resource. A control check function, `guard (q-x)>=0`, is provided to ensure that the "consume" transition is only able to fire (i.e., return "true") if there is at least a quantity `x` of that resource currently available. When some of the resource has been consumed, the quantity remaining is updated. The `''E''` is used as an identifier for the resource, allowing multiple different consumable resources to be represented. The `check` transition uses a bidirectional arc to access the token containing the resource availability and return it without altering it.



(a) Reusable resource

(b) Consumable resource

Figure 4.6: Petri nets for the two main resource types

As stated above, the variables are stored in a separate net which is passed around as a token to be used by all the goals. In the goals net, this is stored in the place named "Variables" with transitions able to reference this net when required. In figure 4.6, the transitions from the variables net with the inscriptions `:check()`

and `:consume("E",x)` can be synchronously fired by transitions in the other nets for checking and consuming resources.

When multiple different consumable resources are used, there can be two types of summary information, depending on the level of detail required. The first provides the detail based on the different resources, while the second gives a single sum for all resources.

The summary information can either be pre-processed (i.e. done off-line) as used here, or potentially produced dynamically by generating Petri nets on-the-fly. Either way, the information produced is the same, and the summary information produced gives the *best case* and *worst case* resource requirements. These are the minimum and maximum resource requirements when taking into account goals or subgoals that have a choice of plans with different summary resource requirements.

The summary information is generated using the tree structure, summing up the requirements starting at the leaves. Where there is a choice of plans, the summaries for those plans are stored with the subgoal to aid the selection between the plans. Renew allows the inclusion of some Java code, so `java.Math` min and max functions from the Java API can be used to identify the best and worst cases. Here, the summaries for the different resources are accumulated together when calculating the summary information so that only a single number is stored for each branch, and the break down is passed on up the tree listing the best case (`bc`) and worst case (`wc`) depending on which branch is chosen. The best case is of course the branch with the lowest resource requirements, while the worst case is the branch with the highest. If some resources are required to be conserved more than others, weights could be added here to indicate an additional cost of using a particular resource, thus favouring the alternatives.

Two forms of the summary information generated are used within the reasoning. The first of these lists how much of each resource will be required in the best case, while the second is a single number totalling the requirements for each of the types of resource. The first of these is used when considering if a goal can be safely adopted, reserving the resources required by the goal. Once the resources are reserved, the reasoning is then concerned with ensuring the best plan options are selected. This can be done through just using a single number representing the total requirements for a branch or alternatively a number indicating which branch

should be selected.

While our approach to the resource summaries here is similar to that of the "necessary" and "possible" summary information used by Thangarajah *et al.* in [Thangarajah *et al.*, 2002], there are some differences in the actual information given. For example, if there were two plans A and B where A used 10 units of resource r1 and B used 5 units of r1 plus 8 units of r2 then the necessary resources listed would be 5 units of r1 as regardless of which plan is used, these resources will always be needed. However, in the approach developed here, the best case summary information would list 10 units of r1, presuming there is no weighting applied, as the sum of the units of resource required by B is greater than that required by A.

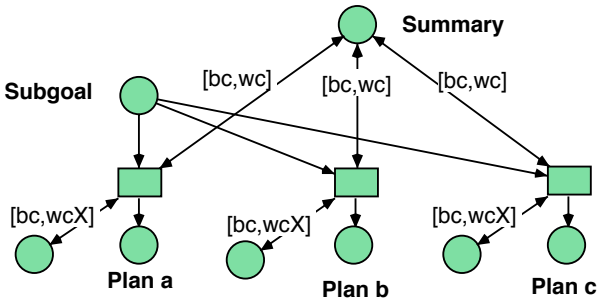


Figure 4.7: Selecting the best plan based on required resources

Figure 4.7 shows how this summary information is added into the goal Petri net to select a branch at a subgoal. When the subgoal is reached, a comparison is performed between the summary of the resource requirements for each of the plan options to the information it has for the best case requirements of the branches. Provided there are no other constraints over the plan selection, it will fire the transition to start the plan with the best case resource requirements.

After all goals have their summary information, the summary information at the root of the tree can then be used by the agent to decide whether it is safe to start acting towards achieving the goal in relation to the amount or resources it currently has available, and any other goals which the agent may be already committed to achieving. Figure 4.8 shows the Petri-net module used by the agent to check the summary information before starting a goal. This summary information is

maintained as a normalised list stating the resource requirements for each type of resource, even if that resource is not required by a given goal.

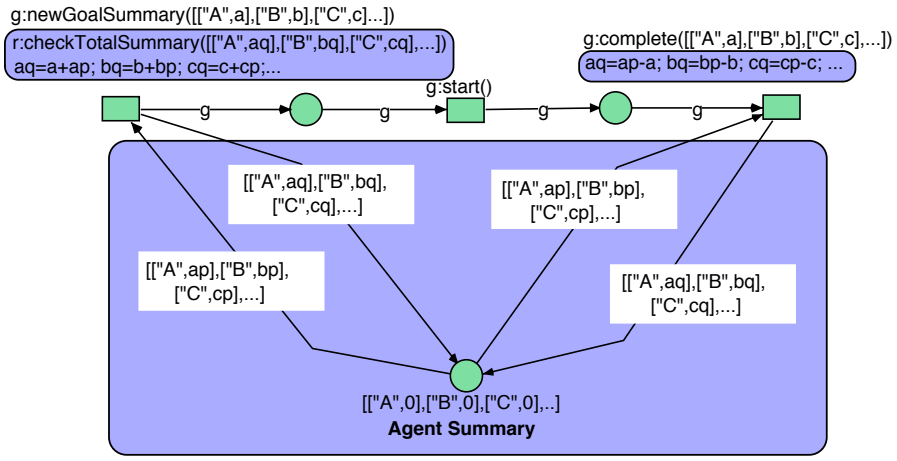


Figure 4.8: Manager module for checking the resource summary information prior to adopting a new goal

The Manager keeps a sum of the summary information for the goals that the agent is committed to achieving, stored in the "Agent Summary" place, so before starting a course of action to achieve a new goal, it checks that there are sufficient resources for the sum of existing goals and the summary from the new goal. If there is, then the goal is adopted and started. When a goal has been achieved, its summary information is removed from the summary for currently executing goals. This is to indicate the total summary information for the goals currently adopted, and does not change as they consume resources whereas the amount of available resources does decrease. As a result, it is possible for the total requirements to be greater than that available at a given point because some resources have already been consumed by the goals that have been started. By removing the summary information of goals that have been completed this can occasionally allow goals that were prevented from starting due to apparent insufficient resource availability to start. While it would be possible to dynamically update the resource requirements of the goals as they are executed, this would introduce additional computational overheads at runtime.

An alternative approach to this may be to record the starting availability of the resources in the Manager net and only compare the summary information to this value. However, if at some future point a maintenance goal was added increasing the amount of available resources in the Variables net, the value in the Manager net would not be automatically updated. This would result in goals being prevented from starting despite there being the resources available for them to safely start.

In order for the total summary information to be checked, additional transitions and places need to be added into the Variables net surrounding all of the different resources. These extensions are shown in the coloured boxes in figure 4.9, having an overseeing transition linked into each of the resources.

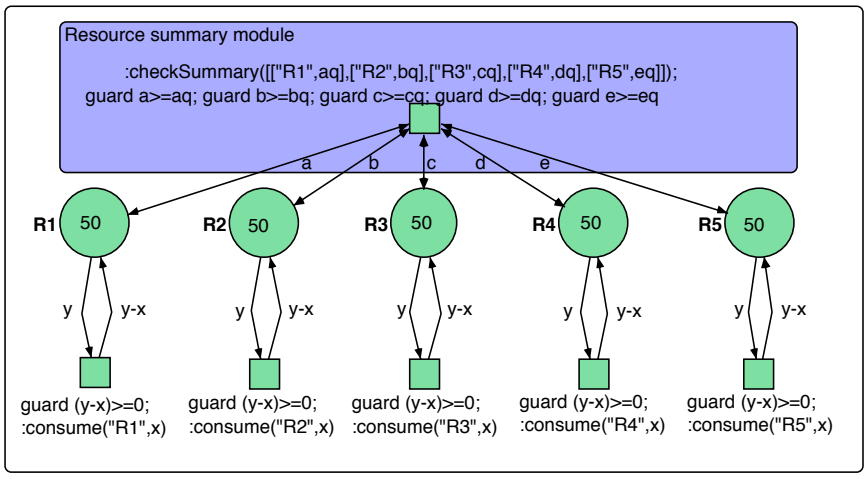


Figure 4.9: Variables net resource summary module

It should be noted here that while this thesis does not extend to reasoning about renewable resources, it is feasible to represent renewable resources in the Petri nets using the construct shown in figure 4.6(a) to allow for the inclusion of reasoning about renewable resources. It is also worth noting here that while maintenance goals designed to restore resources as they are consumed, for example recharging a battery, have not been simulated here, the flexibility in the Petri net model means that should any be added the reasoning about resources is sufficiently robust to handle the regeneration of the consumable resources. This is because the summary information could contain negative numbers indicating the resources that are being produced.

4.4 Modelling Positive Interaction Reasoning

While negative interaction occurs when two goals are setting different values to the same variable (see section 3.4), the positive interaction is modelled by two or more goals assigning the same value to the same variable, as discussed in section 3.3. In essence, this is two or more goals attempting to produce the same effect in the environment, such as a Mars rover going to the same specific location to perform some tests. If the Mars rover is already at the location, it does not need to execute a second plan for a second goal to reach the same location. As a result, the reasoning here needs to check whether the desired effect has already been achieved before allowing the plan to start. If the effect has been achieved then the plan is stopped from executing, so reducing the number of plans executed. Any plans in the sub-tree of this plan are also dropped, potentially having a significant reduction on the number of plans used if one of the plans has a large number of subgoals and plans. As a result, this can speed up the completion and reduce the costs of achieving the goals, particularly if there is a limited amount of resources available.

In the Petri nets, this positive interaction reasoning is handled by a *pre-check* module (Figure 4.10) that first checks whether another plan is about to, or has already, achieved the desired effect of the plan and if not it then fires a transition to indicate that it will be executing a plan to achieve the effect so similar plans for other parallel goals do not need to be executed.

The variables for representing the effects used in the positive reasoning, shown in the top left of figure 4.10 have transitions to set the value and check what it is currently assigned to. The place labelled `var P1` stores the actual value, while the two connected transitions allow other nets to read and alter the current value. The prefix of `P` is used to indicate variables associated with the positive interactions. For the positive reasoning, most of the newly added transitions and places appear in the goals net with the plans that are producing the effects, as shown in the shaded box in the top right of figure 4.10. Before a plan that is going to attempt to achieve an effect starts it checks to see if this effect has been achieved yet. If it has, then the plan is prevented from firing by preventing a token being added

[1]While these are different to those presented in [Shaw and Bordini, 2008] and [Shaw *et al.*, 2008] they still achieve the same results. Changes were made due to restrictions on the arc types recognised in the imported type discussed in section 4.6. This restriction has since been removed.

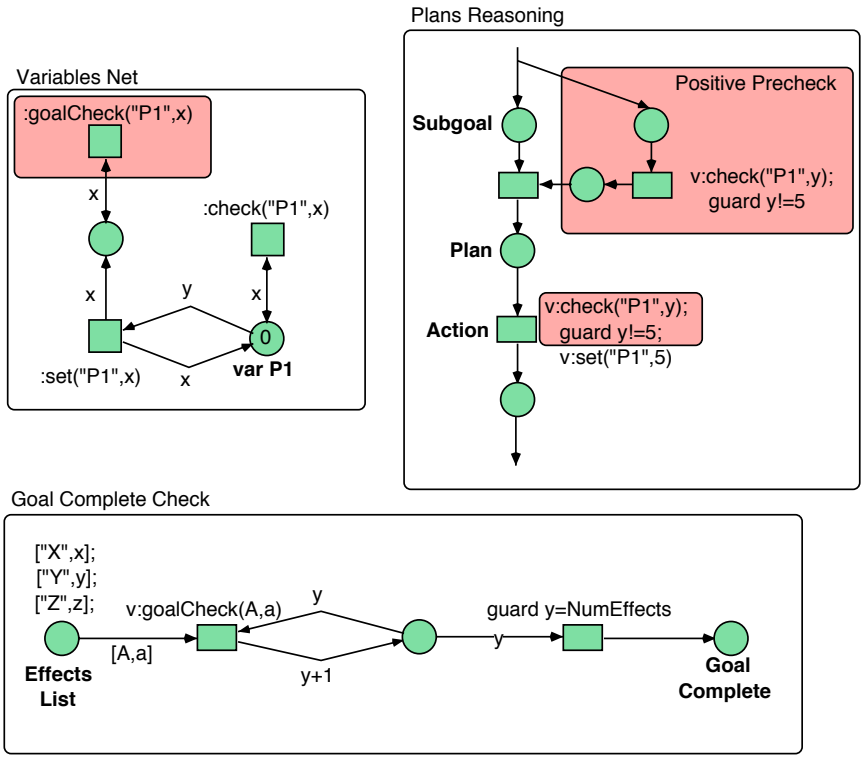


Figure 4.10: Positive module[1]

to a necessary input place for the transition starting the goal. If the effect has not yet been achieved, then the plan is allowed to proceed, checking once again before setting the value that it has not been achieved by another plan running in parallel. If necessary, it would be possible to reserve this effect preventing parallel plans from even starting.

In the positive interaction, as plans are not necessarily completed due to the effects being achieved elsewhere the result is that the approach of checking to see if a goal is achieved based on plans being completed no longer works. An alternative approach of measuring goal completion based on the effects of each goal being achieved is therefore used to see if the goal has been successfully completed. While the tying together of subgoals and plans is still included, an alternative goal-completion check is provided using the effects that should be achieved by the goal. The goal completion is still only counted once, even if all the plans and all the desired effects are achieved. For this to be achieved, an additional "goalCheck"

transition is added to the variable in the variables net (see figure 4.10), linked to a place storing all the effects that have been achieved in that variable. The additional sub-net shown at the bottom of the figure is added to each of the goals, with the list of all effects achieved by that goal. Each of these can then be checked off and when all have been achieved the goal can be considered to be completed.

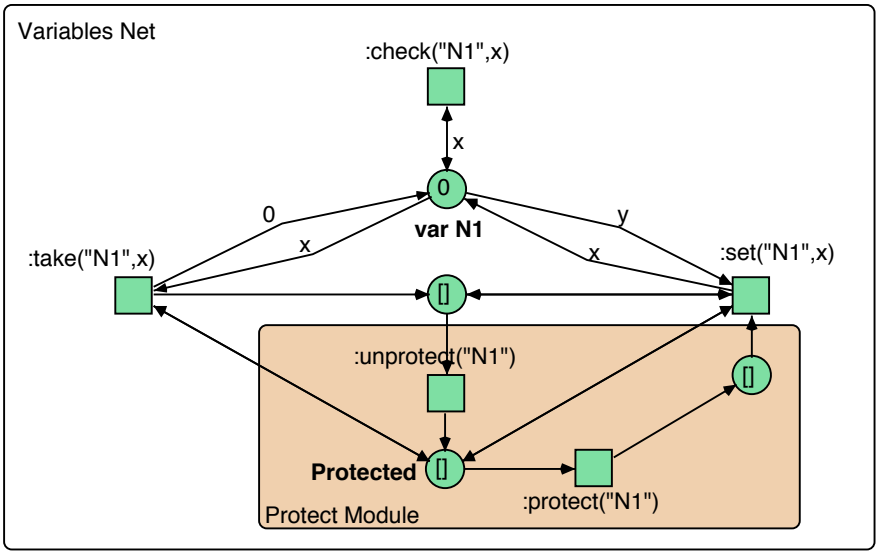
These modules for the positive interaction are automatically included into any plans that are attempting to achieve effects, without performing any reasoning during the generation phase to see if they can potentially interact. This is to ensure that all the reasoning is done by the Petri net itself.

4.5 Modelling Negative Interference Reasoning

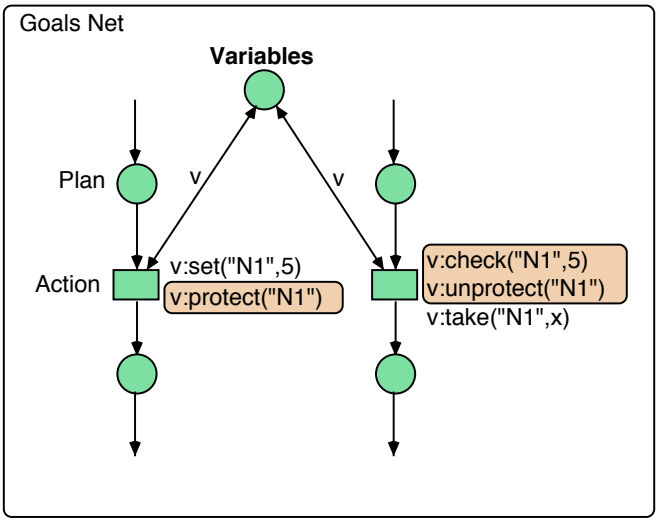
As described in section 3.4, negative interference occurs when two or more goals are referring to the same variable and attempting to change this property in different ways before one of the goals has finished using the variable. As a result, negative interference reasoning has to prevent other goals interfering with any variables that are currently in use, until they are no longer required. Within the Petri net model, this is accomplished by adding in "protection" modules around places where negative interference could potentially occur in order to prevent this interference.

When an agent executes a plan that produces an effect in the environment, and that effect will be required by a later plan, the effect is immediately marked as protected until it is no longer required. This is done by the *protect* module (Figure 4.11) that adds a set of transitions and places to the variables Petri net, and a call from the plan to the new triggering transitions so that when the relevant effect takes place, a transition in the variables Petri net is fired to protect it, then when it is no longer needed another transition in the variables net is fired to *release* the protected effect. If another plan attempts to change something that is currently protected, then it will be stopped and forced to wait until the effects are no longer protected (i.e., until the release transition fires).

Figure 4.11 shows the two areas that are affected by the addition of the negative reasoning. At the top is the representation of the variable, shown by the place `var N1` that starts with the null value of 0 assigned to it. The `N` is used in the examples to indicate variables affected by negative interactions. The transitions directly



(a) Negative interference reasoning modules for Variables net



(b) Negative interference reasoning modules for Goals net

Figure 4.11: Negative modules[1]

surrounding this place allow the variable to be assigned a new value; reset the variable to 0 when the value is removed; and to check the value without changing it to see what it currently contains. The places and transitions contained in the shaded box mark the added components for defining the negative interaction, with a "protected" place containing a token indicating whether the variable can be altered or not. When a token is present in this place, the variable can be altered, however when the protect transition is fired, this token is removed blocking further changes being made to the variable. When the plan has finished with the variable, the take and unprotect transitions are fired to restore a token to the protected place, as can be seen from the plan samples in the lower half of the diagram.

As with the positive interaction reasoning, when generating the Petri nets the relevant protect modules are automatically added to all variables and any plans that are reading or writing variables. No prior reasoning of where the interference will actually occur is performed, nor is any summary information stored about the locations of possible interference.

4.6 Petri net Automated Generation

To automate the production of Petri nets, the Petri Net Markup Language (PNML) was used [Billington *et al.*, 2003]. This is an XML-based interchange format provided for Petri nets that defines the places, transitions and common arcs used in many Petri net editors. This format can then be imported into a Petri net editor or exported to move Petri nets between different editors without loss of data. While the Renew data format itself is XML based, using the PNML format allowed a certain level of abstraction as it is not required that properties such as colour, size and position of the places and transitions are defined, giving a much more streamlined definition of the Petri net.

However, as a result of using the PNML format, the running of the Petri nets has an extra stage before you can get started. Firstly, the PNML files must be imported into a Petri net editor such as Renew [Theorectical Foundations Group, 2006], after which they must be saved with pre-specified names to allow the nets in the different files to interact with one another.

When generating the instances of the Petri net model, it is possible to select

the types of reasoning included, so each type can be considered separately or together. The effects used to cause the negative or positive interactions and the resources that are consumed are only included when the types of reasoning are being considered. For example, the resources being consumed are only included when considering reasoning about consumable resources. It is possible to include the different types of interaction and resource consumption without including the reasoning to produce a base case for comparing the effectiveness of the reasoning against. In this case, during a simulation, the Petri net randomly selects which plans to use with only the tree structure restricting the order of selection. Where there are branches, no reasoning is performed to select which is more appropriate, or stopping plans from interfering with each other and there is no reasoning stopping goals from being started, even when there are not sufficient resources available for the goal to be successfully achieved. This means it is possible to compare the effects of the reasoning against an essentially random equivalent of a Petri net model of the goal-plan tree to consider the costs and benefits associated with the reasoning.

Each instance of the Petri net model is split into three files. There is one file consisting of the goals and plans, another maintaining the variables, and the final managing the starting and monitoring of the goals. When generating the Petri nets in PNML format, the first part to be produced is the Goals file.

Various different approaches were considered for producing the goals. In the experiments performed, each of the top-level goals is assumed to have the same tree structure. However, in real-world applications it is more likely that the goals will each have different tree structures. Due to the assumption, it would be possible to produce a Petri net with a single goal-plan tree represented within it, then create the necessary number of instances of that Petri net for each experiment. The variations in the properties of each goal, such as which variables they were affecting, the values they were assigning to them, and the resources they were consuming could then be passed as parameters to each instance of the net. Unfortunately, this approach leads to a very long and unmanageable list of parameters that need to be passed to each goal, then within the goal they need to be passed to the correct plans. Further drawbacks to this approach include ensuring that all goals defined do indeed have the same structure, and the requirement to still process each of

the goals to generate the list of parameters. If it is known that all goals use the same structure then each goal-plan tree only needs to be traversed to extract the parameters list. However, if a static structure is not assumed, this could cause a large amount of additional pre-processing costs in checking that all the goals do indeed have the same tree structure.

The alternative to this approach is to explicitly define each goal within the Goals Petri net, with all the parameters fixed in place. The trade off here is that as the number of goals increases, so does the size of the file, however the management and construction of the different goals is simplified. The second advantage to this approach is that it allows each of the goals to have a different structure, which is more likely to be the case in a real world example. However, for the purposes of evaluation, each of the goals was given the same structure and it is left to future work to consider the effects of combining a variety of different tree structures together with the different reasoning models.

A variation on the second approach would be to have separate files for each of the different goals, or possibly just the different goal structures, but again this has the drawback of resulting in a large number of files that could cause file management problems, as well as adding some extra complication to the goal management with referring to multiple files. An advantage to this approach is that if in future work a completely new goal was to be added during execution that is different to all existing goals, this would simply require generating a new Petri net Goal file and adding a reference to it in the manager net, rather than attempting to make a large change to the single goal Petri net file of the second approach while it was in use.

While each approach has its advantages and disadvantages, the selected approach to be used was the second approach, with all the goals stored in one large file. In future work, some of these alternatives could be considered to find a way of reducing the size of the Goal file when large numbers of goals are being created, whilst also not generating an excessive number of individual files.

The parsing of the goal-plan tree is done in one iteration through each tree, traversing the trees in a depth first manner, building up the PNML definitions for each of the places and transitions required to fully represent the tree and selected reasoning types. Whilst traversing the trees, lists of the different variables and

resources used are formed in order to produce the Variables Petri net. Best and worst case resource summary information is also calculated to be added at branch points and in the Goal Manager net to reason about which branch to select and whether sufficient resources are available to start a given goal.

The Manager Petri net is the starting point for the simulations. Here the instance of the Variables Petri net is generated and passed as a token to the instance of the Goals Petri net. These two nets are referred to with explicit file names to ensure the Manager is using the correct files. The Manager then controls the start of each of the goals within the Goals Petri net, taking into consideration the resource requirements where this is included in the types of reasoning. The Manager net is also responsible for recording the number of plans used and the goals achieved. Figure 4.12 shows a sample of the Manager net produced when all types of reasoning are included. The long list on the right shows the summary information for the twenty different goals included in this run. The version of the Petri net without the reasoning included, (i.e. the random equivalent), uses the same net for the Manager, however instead of the real summary information, all the resource requirements for each goal are listed as zero. This means that when the check is performed to see if there are sufficient resources, it will always return true as it is only ever checking for zero resources rather than the real amount that is actually required. This check could be removed from the net if necessary to avoid confusion, however it was decided to keep the manager net consistent regardless of whether reasoning was included or not. When the resources are not included, (i.e. for reasoning about only positive or negative interactions), this list is simply the goal ID numbers without the resource summary information. In this case, the consumption of the resources is also not included to ensure that all goals are achievable.

To monitor the performance and collect the results from the simulations of the Petri net model, various parameters need to be collected and stored. These are the goals started, goals achieved and plans used. These are stored in the three large places in figure 4.12.

The two vertical diamonds in the centre are from the manager module for resource reasoning, controlling which goals are started and recording the total

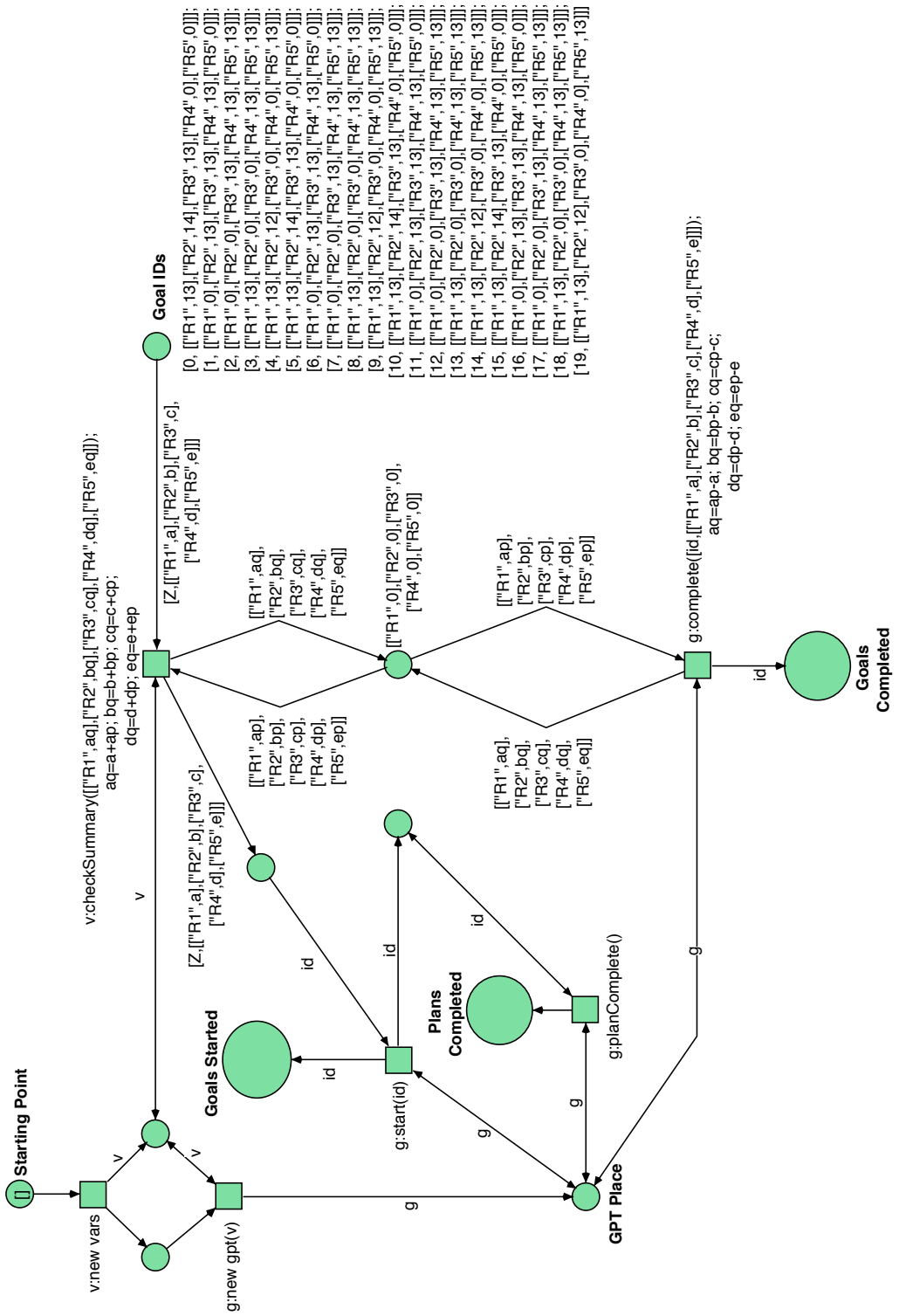


Figure 4.12: A sample of a Manager Petri net model

summary information needed for all currently active goals as described in section 4.3. The transitions within the Goals net marking the completion of plans and top-level goals are inscribed with `:planComplete` and `:goalComplete` references allowing the manager to record when plans have been used and when goals have been completed. These can either be recorded as counters, or the plans can be given IDs that can be appended to a list, specifying the order in which each of the plans for the different goals was used within the simulated execution of the Petri net.

The final Petri net file to be generated is the Variables net. This makes use of the list of variables needed by the goals that was produced whilst generating the main Goals Petri net. Each of these is then included as a sub-net within the Variables net with transitions that can be used to trigger them from the Goals net. Figure 4.13 shows a sample of the Variables Petri net produced with all variables when all types of reasoning are included. In the abstract scenarios, separate variables, representing different factors in the environment, are used for the positive and negative interactions, so the variables on the left are used by the reasoning for positive interaction, and the diamond shaped sub-nets representing further environment variables are used by the reasoning for negative interaction. This is to prevent additional unexpected interactions occurring whilst attempting to evaluate the effectiveness of the reasoning types. However, they could easily be combined and no adverse effects are expected. Results of the experiments performed using the Petri net approach are described in chapter 6. These compare the performance of the Petri net model with and without the reasoning included to the second approach discussed in the next chapter. The Petri net model without the reasoning included is used to provide a base case for evaluating the impact of the reasoning when compared to the performance with an absence of any reasoning.

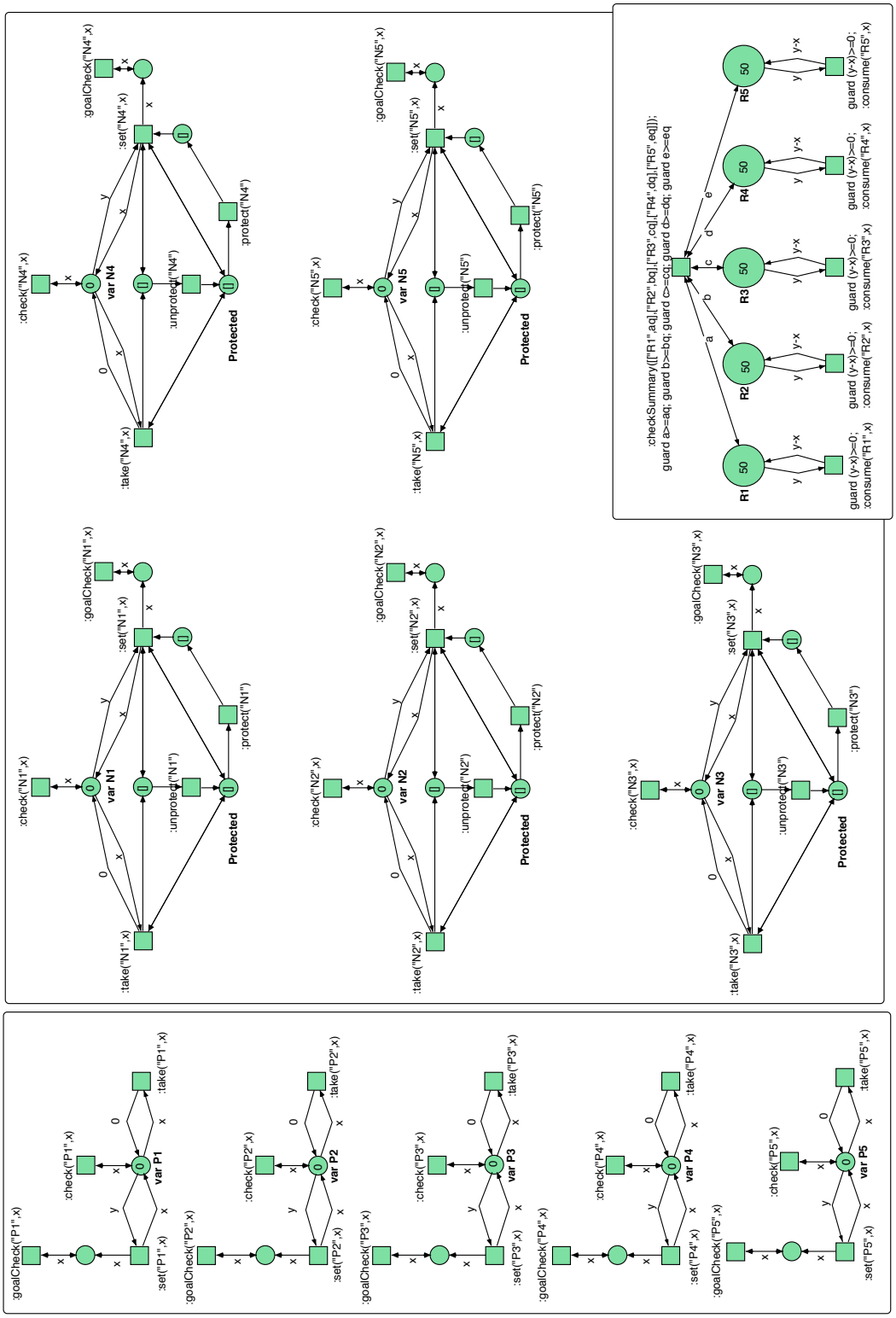


Figure 4.13: A sample of the Variables Petri net model

Chapter 5

Constraint-Based Model

The second approach developed here for reasoning about goals applies constraint satisfaction to find a solution to instances of the goal-plan tree problem. While the Petri net approach applied in chapter 4 provided a natural representation of the agents goal-plan tree into which the reasoning could be added, this approach provides a natural representation of the constraints applied to the agent in the form of resource constraints and interaction constraints.

In this chapter, a description of constraint satisfaction is given in section 5.1, along with an overview of GNU Prolog and the built-in predicates that have been used by this approach. Section 5.2 describes how the goal-plan tree can be represented as a set of constraints, before sections 5.3 – 5.5 describe how to represent and incorporate the three types of reasoning as a series of constraints. This chapter is finished with a description of how the automated generation of this model operates in section 5.6.

5.1 Constraint Satisfaction Problem

Constraint satisfaction attempts to find a solution to a problem consisting of a set of variables, each with a domain of values that can be assigned to the variables. The assignments are restricted by a set of constraints linking the variables that must be satisfied for an assignment to be valid, forming a solution. Each variable can have its own distinct domain of values, or a common domain can be applied

to all variables. This is defined more formally as follows, a Constraint Satisfaction Problem (Constraint Satisfaction Problem (CSP)) consists of a set of variables, each with a non-empty finite domain of values and a set of relations over the variables defining constraints. This can be represented as a 3-tuple $\langle X, D, C \rangle$, where X represents the set of variables $X_1, ..., X_n$, D is a domain of values and C is a set of constraints of the form $\langle t, R \rangle$ where t is a tuple of variables and R is a set of tuples of values of the same size. An evaluation of a CSP gives a mapping of variables onto values from their domains, $v : X \rightarrow D$, with a solution being a complete assignment where every variable is mentioned and all constraints are satisfied [Russell and Norvig, 2003, Chapter 5]. Different queries can be used when evaluating a CSP to access different information, for example in a set of constraints describing family relations a query may just ask who the parents of a given person are, or you could list everyone who was a parent.

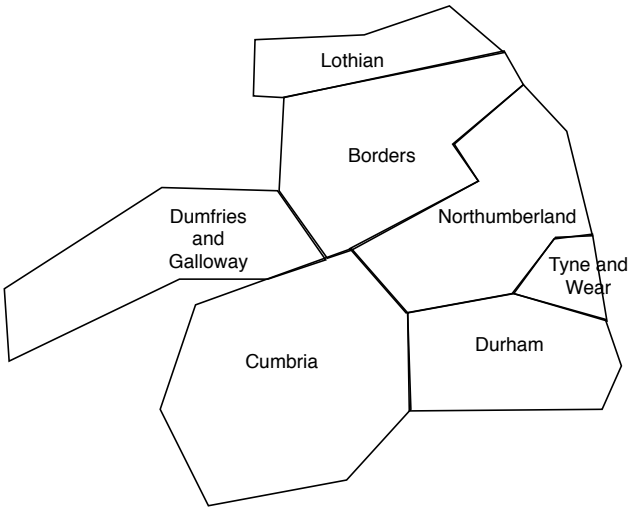


Figure 5.1: Outline map of counties used in map colouring example

A common example to illustrate a CSP is that of map colouring, where the objective is to colour the map without two touching areas having the same colour. On a map of the United Kingdom showing the different counties, the variables in the CSP are the counties, each with the same domain of values, in this case the colours (e.g. $red, green, blue$), to choose from. The constraints are pairs of counties that are connected together, for example based on the outline map shown in figure 5.1,

the constraints $Cumbria \neq Durham \wedge Cumbria \neq Northumberland \wedge Cumbria \neq Borders \wedge Durham \neq Northumberland \wedge Northumberland \neq Borders \wedge Borders \neq Lothian$ can be defined, stating that they cannot be assigned the same colour. An evaluation of this CSP will find multiple possible solutions, for example {*Cumbria = red, Durham = green, Northumberland = blue, Borders = green, Lothian = red*}.

5.1.1 Constraint Logic Programming

Constraint Logic Programming (CLP) combines the constraint satisfaction approach with that of a logic programming language to provide a powerful and sophisticated reasoning and solving engine, an example of this being Prolog. Prolog on its own can be viewed as a simplified constraint satisfaction language, where the constraints are just equalities between terms that are checked by the matching of terms. By adding in additional types of constraints, Prolog can be extended to a CLP language. These additional types of constraints include arithmetic equality and inequality constraints, along with finite domains that are useful when reasoning about CSPs. All of the pure and additional constraints can be used to build up much more sophisticated predicates, however they are unfolded to the basic constraints when evaluating queries [Bratko, 2001, Chapter 14].

The GNU Prolog [Diaz and Codognet, 2000] implementation of the Prolog interpreter, providing constraint solving over finite domains, was used for the development and evaluation of the constraint based model described in this chapter. Others, such as SWI Prolog [Wielemaker, 2003] or ECL^iPS^e [Cheadle, 2008], could also be used with a slight variation in the syntax of the generated constraints. Some preliminary comparisons were performed between these, evaluating the same sets of constraints in each. From these comparisons it was found that for this application the GNU Prolog interpreter proved to be the most effective in terms of time taken and its ability to handle a larger number of goals.

5.1.2 GNU Prolog

GNU Prolog is a freely available application and conforms to the ISO standards for Prolog, while providing some additional features [Diaz, 2009]. Some of the features provided by GNU Prolog include more than 300 built-in predicates, global variables and a constraint solver amongst others. The constraint solver provides an efficient finite domain solver when compared to commercial constraint solvers, integrating finite domain variables into the Prolog environment, with a lot of predefined constraints including more than 50 finite domain constraints and predicates. If necessary, the user can extend this by defining their own new constraints. A summary of the predicates, constraints and notation used in the development of this model are given below. Where additional predicates are defined for common operations such as finding the intersection of two sets, these are based on the techniques described in [Bratko, 2001].

CSPs are generally solved using some form of search or inference algorithms [Tsang, 1993]. These are typically based on backtracking, constraint propagation and local search techniques. The backtracking algorithm consists of two phases, forwards and backwards. When going forwards, the algorithm works through the variables in sequence assigning a value from their domain that is consistent with all the constraints and values assigned to previous variables. When the algorithm reaches a variable for which there is no valid assignment, it then backtracks to the previous variable, changes the value and attempts to move forwards again [Dechter, 2003]. The points to which the algorithm backtracks are known as choice points, where the solver made a choice of value to assign to a variable. The map colouring example given above is simple enough that the first available value for each county will give a valid solution. In a more complex example, backtracking would be required to find a valid solution.

A technique called constraint propagation is used to reduce the size of the problem into something that is smaller and simpler to solve. This takes into consideration local consistency, where consistent solutions for a subset of the variables are maintained. Constraints are propagated using node and arc consistency and is the approach used by the solver in GNU Prolog.

Node consistency applies constraints to the domains of variables so that the constraints can then be discarded, for example, a variable V with domain $\{1, 2, 3, 4\}$ and a constraint $V < 3$ will have its domain reduced to $\{1, 2\}$ and the constraint removed to simplify the constraint checking of the final solution.

Arc consistency considers the constraints between two variables. For example, suppose variables X and Y each have the domain $\{1, 2, 3, 4\}$ and a constraint between them $X < Y$. The value 4 can be removed from the domain of X as this value will never satisfy the constraint. Similarly, the value 1 can be removed from the domain of Y. However, unlike in node consistency, the constraint cannot be discarded as it would still be possible to assign values to the two variables that would not satisfy the constraint. As each variable's domain is updated, all the other variables need to be reconsidered to ensure all arcs remain consistent, propagating the updates through all the variables. Path consistency extends this notion by considering variables along arcs that form paths connecting all the variables together.

It is possible to draw a tree of a CSP to represent its search space, where each node in the tree represents a choice made, with the branches from that point representing the choices available. For example, the first set of branches in the map colouring problem described above would represent the counties available for selection. Each of the counties would then have a branch for each colour. After a colour had been assigned to a county, another county would be selected, with the list of colours available from there, as illustrated in figure 5.2. In the diagram, only the first branch has been expanded in a depth-first search to illustrate the branching in the search space. The depth of the tree is fixed based on the number of variables in the problem and in a fully expanded tree the leaves at the maximum depth would represent the end points of paths containing valid solutions.

Local searching starts with a complete but probably invalid assignment and searches the local search space in order to improve the assignment. There is a risk that a solution will never be found with this approach, even if a valid solution does exist. This is due to the search being restricted to a small area of the full search space.

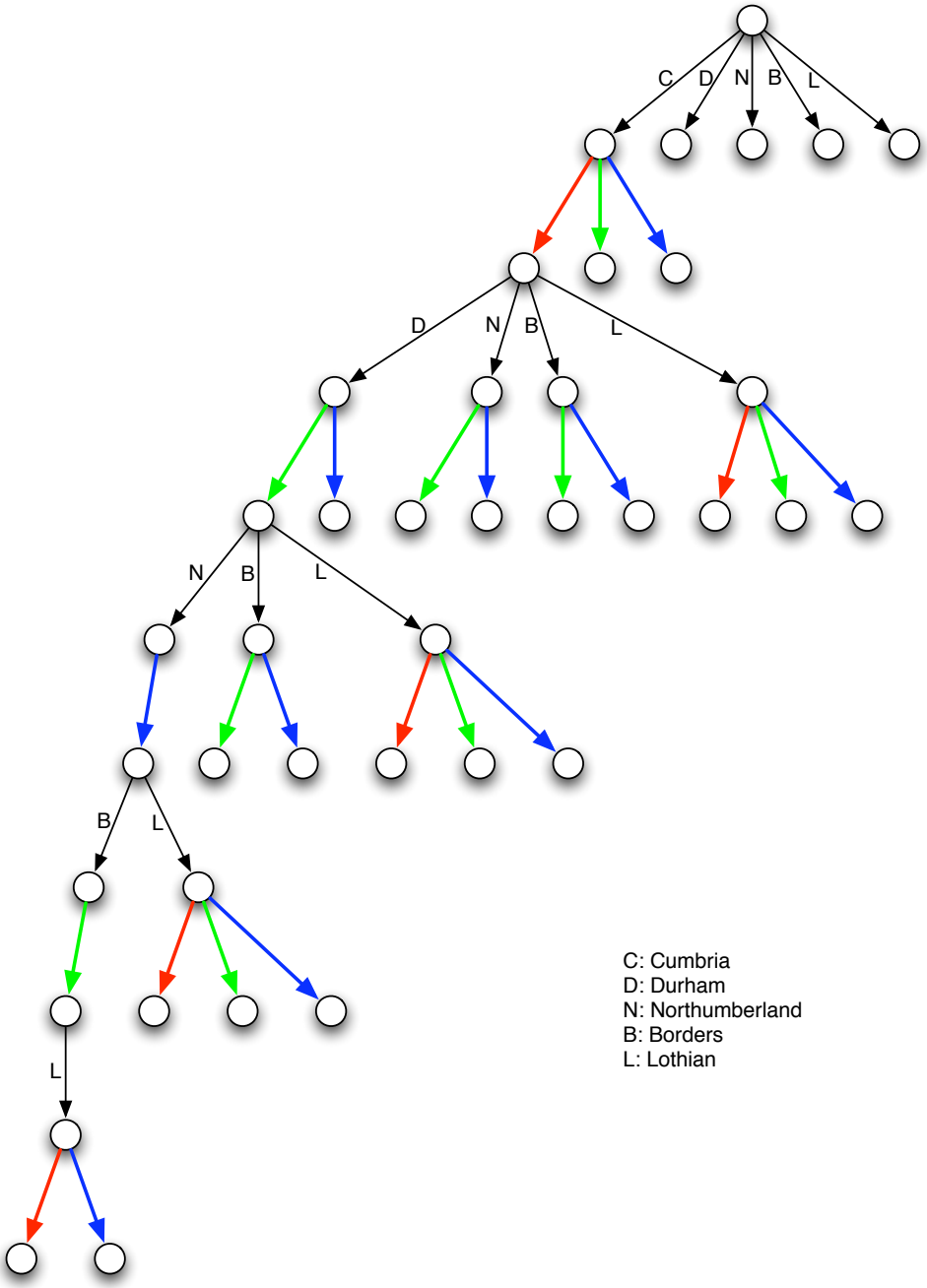


Figure 5.2: Search tree for map colouring problem

5.1.3 GNU Prolog Notation

In Prolog, facts such as "Socrates is male" and "all men are mortal" can be represented as shown below. In GNU Prolog, all descriptions for a given predicate must be consecutive unless they have been defined to be discontiguous using the predicate `discontiguous(Predicate)`, which removes the restriction requiring all the clauses defining `Predicate` to be consecutive within a source file. This was used when defining the goals and plans (described in section 5.2) allowing them to be listed as they were encountered within a goal-plan tree, rather than being required to list all the goals followed by all the plans separately.

The following is an example of defining facts based on family relations within Prolog:

```
parent(raymond, patricia).
parent(valerie, patricia).
parent(frances, valerie).
parent(leonard, valerie).
...
```

Questions can be asked about the facts defined, or about the rules and relations defined in the Prolog source. Four example queries of the facts above are shown here:

```
parent(valerie, patricia).
parent(X, patricia).
parent(X,_).
parent(X,Y), parent(Y, patricia).
```

The first query asks if Valerie is a parent of Patricia, while the second is slightly more general asking who are Patricia's parents. The result of the first would simply be `yes` as there is a fact matching the query, while in the second we are asking Prolog to unify, i.e. associate, the variable X with all the parent facts where the second term matches the atom `patricia`. The first result will therefore be `X = raymond` and the next answer `X = valerie`. Unless you tell Prolog to give you all answers, it will only list one at a time, waiting for you to request the next or finish. The source

is evaluated from top down, so while the relation `parent(raymond, patricia).` appears before `parent(valerie, patricia).`, the answer `X = raymond` will always be given before `X = valerie`.

The query on the third line shown above is only interested in finding out who is a parent. In Prolog, the _ character indicates that we are not interested in that term, so this will give Raymond, Valerie, Frances and Leonard as answers. Patricia is not listed as the first term for any of the parent relations, so is not given in this answer. Finally, the last query on line 4 is a compound query asking who the grandparents of Patricia are. The first solution will give `X = frances, Y = valerie` and a second solution of `X = leonard, Y = valerie`. A rule for this could be written in the source file saving the user from having to enter the compound query each time they wanted to ask the same question, as shown here:

```
grandparent(X, Z):-
    parent(X, Y),
    parent(Y, Z).
```

The symbol `:-` separates the head of the clause or rule from the body. In order for the head to be true, all of the body must also evaluate to true, in this case finding a valid unification of each of the variables in the clause.

Prolog provides a selection of control constructs for use within clauses, including infix operators for conjunction and disjunction, along with a construct for if-then-else control. When evaluating these constructs, `true` will always succeed, while `fail` will force the query to backtrack to the last "choice-point" in order to try a different assignment of terms. If required, the "cut" construct represented by '`!`' can be used to remove all choice-points created up to that point. This means that if the solver needs to backtrack, this is the last point to which it will back track as any choice-points before this point will have been removed.

Conjunctions are represented with the '`,`' operator and disjunction by the '`;`' operator. The if-then construct is represented by a '`->`' operator and a '`;`' can be added to represent the else part of the statement, for example:

```
Task1 -> Task2 ; Task3.
```

This is evaluated by executing the first query, `Goal1`, and if it is successful then the second query is executed, `Goal2`. When the else part is included, in this example `Goal3`, this is only executed if the first query fails or if the evaluation backtracks to this point.

Lists are represented in Prolog using square brackets, where `[H | T]` can be used to obtain the head and tail of a list. `H` is unified with the first element in the list, while `T` is unified with the tail of the list, i.e. the original list with the first element removed. A variety of predicates are supplied for manipulating lists, including `append`, `member` and `reverse` amongst others.

It is often useful to be able to represent pairs of terms or variables. While this could be done with a list of two elements, a simpler notation is provided in Prolog using the form `V1/V2`.

One of the additional features provided in GNU Prolog outside the Prolog ISO standard is the availability of global variables. These can be accessed using `g_assign(GVarName, Value)` and `g_read(GVarName,Value)`. Related to this are dynamic clauses that can contain entire relations and can be asserted and retracted dynamically in Prolog. The predicates `asserta(Clause)`, `retract(Clause)` and `retractall(Head)` can be used to add and remove these clauses. Any predefined clauses in the Prolog source need to be declared as dynamic using `dynamic(Head)` in order to be able to assert and retract them. This can be useful to keep track of values for example a Fibonacci series can be generated with the first n numbers in the series being asserted as facts that can then be queried. In the approach developed here, this is used to monitor resource availability of the different consumable resources, as described in section 5.3.

When performing a query, it is sometimes useful to obtain all the possible solutions in one go. In order to achieve this, Prolog provides a `findall(Template, Query, Instances)` predicate to automate the process, building up the list `Instances` of all the solutions for `Query`. `Template` provides the format for the results that are added to the list, for example if we wanted to find all grandparents we could use the query `findall(X,grandparent(X, Y),GP)`.

Finite Domains The finite domain solver incorporated into GNU Prolog is based on the clp(FD) solver [Codognet and Diaz, 1996]. The new constraints added

include arithmetic, boolean, reified and symbolic constraints. These constraints are solved using arc-consistency propagation techniques. In addition to the new constraints included with the finite domain extension, a new type of variable is also added. This is a finite domain variable that has an integer data type that can be limited to ranges or specific values. The variable can then only be assigned a value from its domain. These domains can be set using `fd_domain(VarsList, Lower, Upper)` to give each element in the `VarsList` a domain of values from *Lower* to *Upper*. Alternatively a list of values can be given in place of the lower and upper bounds to specify a non-consecutive range of values.

The basic arithmetic operators are represented simply by using the standard symbols, for example, '`+`' for summing integer values of two variables, while the arithmetic constraints in the finite domain solver are represented by prefixing and suffixing the operator with a # symbol. This can either be done with just the prefixed # to only apply the constraint to the bounds of the domain of the variables (e.g. `#=<` meaning less than or equal to), or if both are used the full domain of the variables are updated (e.g. `#=<#`). The use of the partial update is generally more efficient for arithmetic constraints as less propagation is required.

After all the constraints have been defined, the final step used in the finite domain solving is that of assigning specific values to each of the variables in order to satisfy all of the constraints. This is done by using the predicate `fd_labeling(VarsList, [variable_method(Heuristic)])`. The second argument is optional, as the default standard heuristic, which simply works through the list starting from the left applying values to each as they are considered, is used if no heuristic is specified.

Heuristics When evaluating the CSP, different heuristics can be used to vary the order in which variables are mapped to values when considering the constraints upon them. Some of these heuristics include *standard*, *most constrained* and *maximum regret*. The *standard* heuristic simply starts at the leftmost variable or the first variable in the list and works the way through the list assigning the first available consistent value, while the most constrained and maximum regret heuristics apply some ordering to the list of variables before starting to assign values to them. As suggested by the name, the *most constrained* heuristic orders the list such that

those variables with the smallest number of elements in their domain are at the start of the list. Selecting these elements first will reduce the amount of backtracking required to find a satisfying assignment for all the variables, or identify sooner if no valid solution exists. When multiple variables have the same number of values in their domains then the variable appearing in the greatest number of constraints is selected first.

The *maximum regret* heuristic orders the variables based on the difference between the smallest value and the next value of its domain, for example if there were two variables X and Y where X had the domain $\{1, 3, 7\}$ and Y had the domain $\{1, 5, 6\}$ then the solver would select Y first as there is a greater distance between the first and second values in the domain. As with the most constrained heuristic, if there is a tie between two or more variables, the variable appearing in the greatest number of constraints is selected first.

An example of the map colouring CSP described above, using the outline counties map in figure 5.1, is presented here in GNU Prolog to show how some of the predicates described above are used and to demonstrate the three heuristics discussed above. In this example, numbers have been used instead of words to identify the colours, i.e. $0 = red, 1 = green, 2 = blue$. Text placed after % symbols are comments.

```
uk(V):-
                % V: List of variables representing counties
    V=[Cumbria, Durham, TyneAndWear, Northumberland,
       Borders, DumfriesAndGalloway, Lothian],
                % set domain of each variable in V to {0,1,2}
    fd_domain(V,0,2),
                % define constraints between counties
    Cumbria #\= Durham,
    Cumbria #\= Northumberland,
    Cumbria #\= Borders,
    Cumbria #\= DumfriesAndGalloway,
    Durham #\= Northumberland,
    Durham #\= TyneAndWear,
```

```
Northumberland #\= TyneAndWear,
Northumberland #\= Borders,
Borders #\= DumfriesAndGalloway,
Borders #\= Lothian,
            % assign values to each variable in V
fd_labeling(V).
```

When this is evaluated with the default *standard* heuristic, the first result returned is $V = [0, 1, 0, 2, 1, 2, 0]$, which is the list of values assigned to each of the variables in V, i.e. {*Cumbria = 0, Durham = 1, TyneAndWear = 0, Northumberland = 2, Borders = 1, DumfriesAndGalloway = 2, Lothian = 0*}. This was found by assigning the first available value to the first element in the list and working through. Again, this example was simple enough that backtracking was not necessary, so the first available values were sufficient for each variable.

Replacing the final predicate for labelling the variables, `fd_labeling(V)`, with `fd_labeling(V, [variable_method( most_constrained)])` changes the heuristic used to order the list of variables from the default to the most constrained heuristic. As all variables initially have the same sized domains the ordering is based on the number of constraints each variable appears in. In this example the list is dynamically reordered to: *[Cumbria, Northumberland, Borders, Durham, TyneAndWear, DumfriesAndGalloway, Lothian]*. Cumbria is assigned the value 0, removing this value from the domains of Durham, Northumberland, Borders and DumfriesAndGalloway. The list is again reordered to indicate the most constrained variables, now appearing as: *[Cumbria, Northumberland, Borders, Durham, DumfriesAndGalloway, TyneAndWear, Lothian]*. The next unassigned variable, Northumberland, is selected with the first available value, 1, being assigned from its domain. This removes the value 1 from the domains of Durham, TyneAndWear and Borders. Both Borders and Durham are each constrained to one value in their domains, however as Borders appears in more constraints than Durham the list remains unchanged at this iteration. The next two variables assigned are Borders and Durham, each being assigned the value 2 from their domain and constricting DumfriesAndGalloway and TyneAndWear to one value each, these being 1 and 0 respectively. The final variable Lothian still has a choice

of two values so the first of these, 0 is selected to complete the solution. Giving the solution in the original ordering of the variables, the final solution returned was $[0, 2, 0, 1, 2, 1, 0]$.

Finally, if evaluating this example with the maximum regret heuristic, written `fd_labeling(V, [variable_method(max_regret)])`, the evaluation starts with the same ordering as for the most constrained heuristic. After assigning Cumbria with 0 all variables still have the same level of 'regret', however DumfriesAndGalloway is again more constrained than TyneAndWear so steps just ahead of it as it did when using the most constrained heuristic. Northumbria is again assigned the value 1 from its domain. This time however, when the value is removed from TyneAndWears domain, the 'regret' level of this variable is the highest out of all the variables, with a domain of $[0, 2]$, so it is moved up the list ahead of Borders and is selected next. The value 0 is assigned to TyneAndWear, followed by 2 to Borders and Durham again. Next DumfriesAndGalloway is assigned 1, finishing with 0 assigned to Lothian. While the final ordering is the same as that produced by the most constrained heuristic in this example, the order in which the variable were assigned values was slightly different. In a larger more complex example these could have produced different solutions.

For the purposes of comparison, each of these heuristics listed here will be used when evaluating the effectiveness of the constraint-based approach to see if one heuristic was more suitable to this problem domain than another.

5.2 Modelling the Goal-Plan Tree

In chapter 4, we presented the Petri net model developed for representing the goal-plan tree problem. In the remainder of this chapter, we present our second approach, developed here, for representing and reasoning about this problem. This uses constraints in GNU Prolog to represent the goal-plan trees and reason about the same three types of reasoning as that done in the Petri net model. The same types of summary information as used for reasoning about consumable resources in the previous chapter are used in this reasoning and the reasoning about positive and negative interactions has again been developed in such a way as to avoid the

use of any summary information. Each of these types of reasoning are discussed in sections 5.3 – 5.5. Firstly, we present the approach to simply representing the goal-plan tree using constraints in GNU Prolog.

The idea surrounding the model used for representing the goal-plan tree problem as a set of constraints is to find an ordering of the plans for all of the goals such that all the goals adopted are achieved and as many goals as possible are adopted.

To start with, the plans and goals are both defined as facts using the name `node` within Prolog, with the plans being represented by 5-tuples $\langle Pl, S, Pr, E, R\rangle$ where Pl is a unique identifier for each plan; S is the list of subgoals for achieving the plan; Pr is a list of preconditions and E is a list of effects caused by the plan; R is a list of pairs showing the resource requirements for the different resources that a plan uses. The plans at the bottom of the goal-plan tree that form the leaves of the tree will not have any subgoals listed in S, and not all plans will have preconditions, effects or resource requirements.

As in the Petri net model, a series of "variables" are used to represent resources and the effects on the environment. The resources make use of dynamic facts that can be retracted and asserted with updated values as the resources are consumed, (e.g. `resource(r1,50)`. The effects on the environment use a similar form to those used in the Petri net where a series of places, representing different attributes within the environment that can be modified, stored values indicating the current state of a given attribute within the environment. In Prolog, the effects on the environment are simply represented as pairs consisting of the attribute identifier and the value representing its current state (eg. `e1/7`). As is described in sections 5.4 and 5.5 these are used to identify plans that can either be safely merged or that could interfere so need to be scheduled accordngly.

In the plan definitions, the preconditions, effects and resources are all represented as pairs of values, for example $r1/5$ represents the requirement of 5 units of resource $r1$. The preconditions and effects are represented in a similar way with $e1/7$ stating that the plan changes the variable representing the environment factor e1 to have the value 7.

Goals and subgoals require less details so they are simply represented as 2-tuples, $\langle G, P\rangle$, where the G is a unique identifier for the goal or subgoal and P

is a non-empty list of plans that can be used to achieve G. The following Prolog sample from the goal's definition shows a top-level goal node and a plan node that achieves this goal, using itself 1 unit of resource r1 and causing the effect of assigning the value 7 to variable e3, while having no preconditions required for it to start.

```
node(aag48,[aap50]).                                % Goal node
node(aap50,[aasg22,aasg47],[],[e3/7],[r1/1]).       % Plan node
```

To improve the flow of the goals definitions, the definitions for the goals, subgoals and plans are all interspersed rather than writing out the definitions for all the plans followed by all the goals and subgoals. As a result, Prolog requires that the node predicates are defined as discontiguous so this is managed by the predicates `discontiguous(node/2).` and `discontiguous(node/5).` being placed before the start of the node definitions.

These node definitions allow a set of predicates to be defined for traversing the tree, for example generating a list of branch options, and sub-trees. This also allows the definition of predicates to remove branches such as when choosing between multiple plans for a given subgoal or goal, or when dropping plans as a result of positive interaction.

In order to reason about the tree structure, various predicates are defined to query the definitions of the goal-plan tree. These include listing all the plans in the sub-tree of a goal or plan, finding all the plan options for achieving a goal or subgoal and querying plan hierarchy within the goal-plan tree.

Where there is a choice of plans to achieve a goal or subgoal, only one of these needs to be used in order for the goal to be successful. The surplus plans can therefore be removed from consideration, reducing the number of plans that need to be considered later on. Where resource reasoning is included in the reasoning being performed, the choice of plan selected will be based on the resource costs of the different options, keeping the plan with the lowest resource requirements. To ensure that the unnecessary plans are removed from consideration their plan definitions are retracted. In doing so, the sub-tree of the plan also needs to be retracted where the plan contains subgoals. This is illustrated in figure 5.3 where

the plan and its sub-tree inside the dashed line is being retracted in preference of the alternative plan for achieving the subgoal.

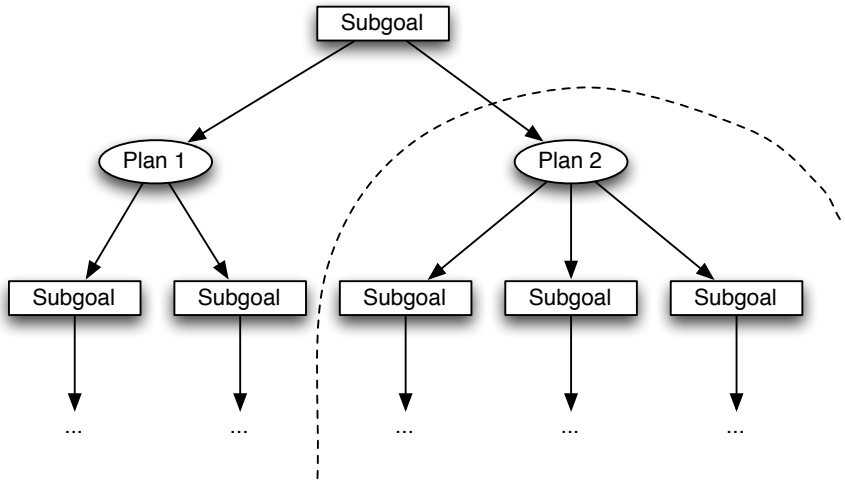


Figure 5.3: Removal of surplus sub-trees where there is a choice of plans

In Prolog, this is defined as a series of predicates to "strip" the tree of the branch options:

```
branchOptions:-
    findall(O,option(_,O),All),
    branchStrip(All).
```

This first predicate uses the `option(Goal,OptionList)` predicate to generate a list of all the sets of options for subgoal branches. `O` is a list of plans from which just one plan needs to be selected so the variable `A`, the result of the `findall` equates to a list of plan lists. Each of these lists of plans then needs to be considered, selecting one plan to keep and the remainder to retract. By default, the plan that is kept is the first plan in the list, however when resource reasoning is incorporated, the summary resource requirements for each branch is considered so the plan with the lowest summary resource requirements is kept.

```
branchStrip([]).
branchStrip([[H | T2] | T]):-
    rmBranch(T2),
    branchStrip(T).

rmBranch([]).
rmBranch([P|T]):-
    strip(P),
    rmBranch(T).
```

When removing plans, it is important to remember to remove the sub-tree formed from any subgoals that were required by the plan. This is handled by a final recursive predicate to iterate through the list ensuring each of the members of the sub-tree are removed. As it is possible for plans within the subtree of an optional plan to also contain branches, it is feasible for plans and subgoals to have already been removed. To prevent this from causing the retraction to fail a disjunction finishing with `true` is included as shown below.

```
strip(P):-
    subtree(P,T),!,
    stripTree(T),
    retract(node(P,_,_,_,_)).

stripTree([]).
stripTree([H|T]):-
    (((retract(node(H,_,_,_,_))); retract(node(H,_))); true),
    stripTree(T).
```

An evaluation of the CSP gives each plan that is considered a number that can be used to sequence the plans. A global finite domain variable is created for each of the plans to store the plans domain of values, ranging from 0 to the number of plans. A solution is a valid sequence where the goals adopted would be achieved if the plans were executed in the order specified by the evaluation. A tree scheduling predicate, `treeScheduler` shown below, is applied to the plan variables to ensure the tree structure is maintained when considering the order in which to execute plans, forming the basis of any scheduling over the plans. This includes preconditions and effects of plans between different branches within a tree to ensure a plan is not scheduled to execute before the plan producing the necessary preconditions has been scheduled to execute.

```
treeScheduler([]).
treeScheduler([[P1,P2]|T]):-
    g_read(P1,I),
    g_read(P2,J),
    I#<#J,
    g_assign(P1,I),
    g_assign(P2,J),
    treeScheduler(T).
```

In many cases, the ordering between subsets of the plans is not important as they will not affect each other in any way so these plans can safely be given the same sequence number. When executing the plans, this could be seen as either

executing them in parallel or executing them in sets, such that all the plans with sequence number 1 are executed before those with sequence number 2, and so on and so forth. By not specifying an exact ordering to the plans, the agent is able to maintain a lot of its autonomy when selecting which plan to execute next. Essentially the "ordering" of plans indicates to the agent which plans are safe to execute together, grouping them into "safe" sets. Provided the agent completes all the plans within one group before moving onto the next, there should be no interference between the various goals. In the worst case, where there was a lot of interference between all of the goals, each plan could be assigned a unique number from their domain of values, specifying an exact ordering in which the plans must be executed for the agent to be successful.

When searching for valid solutions to goal-plan tree problem, the query is directed from the `reasoning` predicate shown below. When a solution is found, each of the parameters in the head of the predicate is unified with part of the solution or details about the solution for evaluation purposes. This includes counting the number of plans used, the number of goals achieved and the time taken for the solution to be found. The Prolog predicate `real_time(Time)` is used to obtain start and end timings for the evaluation of the model.

```
reasoning(Schedule, Plans, PlanCount, TimeTake,
    GoalsSet, GoalsAchieved):-
                            % start timing the reasoning
    real_time(Start),

    findall(G,root(G),Goals),
    length(Goals, GoalsSet),

    branchOptions,
                            % positive interaction reasoning
    findall([Pa,Pb],pos(Pa,Pb),Merge),
    posScheduler(Merge),
                            % resource reasoning
    branchList(Goals,SumList),
```

```
sort(SumList,SortedSumList),
resReasoning(SortedSumList),

findall(P,node(P,_,_,_,_),Plans),
length(Plans,PlanCount),
varSetup(Plans,PlanCount),

findall([Px,Py],tree(Px,Py),A),
reverse(A,A2),
treeScheduler(A2),
                          % negative interference reasoning
findall([Pc,Pd,Pe],neg(Pc,Pd,Pe),Neg),
negScheduler(Neg),

varResult(Plans,Schedule),
fd_labeling(Schedule,[variable_method(standard)]),
                          % reasoning finished
real_time(End),

TimeTaken#=End-Start,
findall(G2,root(G2),Goals2),
length(Goals2,GoalsAchieved).
```

The first step in the predicate unifies the variable `Goals` with a list of all the top-level goals. The length of this list is queried to identify how many goals have been defined at the start. When reasoning about consumable resources, it is likely that not all goals will be achieved, so a repeat of this is performed to count the number of goals after the actual reasoning and scheduling components of this predicate have been completed. Once the list of goals has been unified, the reduction of the goal-plan trees can start by removing the branch options as described above. If reasoning about positive interactions is included (section 5.4), then this is inserted after the branch options have been removed. Similarly, if the consumable resource reasoning is incorporated into the constraints (section 5.3),

the main reasoning concerning which goals can safely be adopted is inserted after the branch options and after the positive interaction reasoning where both are included.

Once all the plans have been removed that are surplus to requirements either because of branch options, positive interactions or limited resources restricting the number of goals that can be adopted, the finite domain variables for each of the remaining plans are asserted as global variables. This is contained within a `varSetup` predicate which iterates through the list of all the remaining plans asserting the global variables with the domain ranging from 0 to the number of plans now being considered, i.e. the length of the list of plans. After this has been successfully completed, it is then possible to start applying the constraints that restrict the assignment of the values from the domains to the variables. This starts with the scheduling based on the tree structure and finishes with the negative interference reasoning (section 5.5), when this is incorporated into the types of reasoning being performed. At this point, the labelling of values to variables is performed, so the `varResult` predicate collects all of the finite domain variables back into a list which is passed to the finite domain labelling predicate. In this predicate, the heuristic to be used by the solver for labelling the variables is specified. In the extract below, the default heuristic is applied, however the *most constrained* and *maximum regret* heuristics are also applied in the evaluation of this approach (see chapter 6). Once the labelling has been completed, the reasoning process has finished so the variable `End` is unified with the system time to calculate how long the solver has taken to find a solution. A final goal count is performed when the reasoning about resources is included to count the number of goals the agent was able to adopt and successfully achieve.

While this design achieves the objectives of representing and reasoning about the goal-plan tree it may be possible to optimise some of the constraints in order to improve their efficiency, thereby reducing the length of time taken for a solution to be found.

The definition of the constraints used in this approach are split across three files. The first of these files contains the discontiguous definitions of all the goal and plan nodes for each of the goals, while the second contains all the constraints for reasoning about the goal-plan trees. The final file, contains some utility predicates

that were not included in the built-in predicates for GNU Prolog. These consist of definitions for finding the union and intersections of two lists along with identifying if two unordered lists are equivalent.

As with the Petri net approach, the constraint based approach has again been designed using a modular approach. This means that when generating the constraints for a given goal-plan tree it is possible to restrict the types of reasoning included to just cover one or more of the reasoning types. If all the types of reasoning were omitted, the constraints produced would only model the tree structure, producing an ordering based on this alone with no constraints preventing it from scheduling plans after all the resources have been consumed or causing plans to fail due to interference. The solutions given would be solutions to problems without any limitations on resources or interactions between goals, unlike the results generated by the random version of the Petri net model where it is possible to include the resource consumption and goal interaction without having to include the reasoning. As a result, experiments using this model are only concerned with the inclusion of at least one of the three types of reasoning.

In order to model the goal-plan tree problem as constraints, a set of predicate functions has been defined to represent each of the different forms of reasoning. Each of these predicates can then be used to form the additional reasoning constraints over the domains of variables as shown in the subsequent sections.

5.3 Modelling Consumable Resource Reasoning

The reasoning described here is again limited to that of consumable resources rather than reusable resources. As with the Petri net model, the purpose of the reasoning is to restrict the number of goals adopted to those that can be achieved with the amount of consumable resources available and to endeavour to make the best use of those resources through the careful selection of plans when there is a choice between which plan to use in order to achieve the desired result. As with the Petri net model, the reasoning about consumable resources again makes use of a small amount of generated summary information to perform this reasoning.

As described in the section above, the resource requirements for each plan are represented by a list of pairs consisting of resource type and quantity required.

The total available resources for each type are each defined using a `resource` predicate as shown below. This predicate is defined to be dynamic so that when reasoning about resources the quantity available can be updated by retracting and reasserting the predicate with the new quantity. While it would have been possible to use global variables to achieve the same result, this method was used to improve the readability of queries over the quantity of resources available.

```
resource(r1,50).
```

The first part of the resource reasoning is incorporated into the constraint reasoning for the selection between lists of plan options for achieving a goal or subgoal. For each of the plans listed as being an option, a summary of the resource requirements for the sub-tree with the plan at its root is generated. At this point, a single number for all the resource quantities required regardless of resource type is used to decide which plan to use. It is possible to extend the reasoning here to incorporate weightings into the summation of resource requirements in order to indicate preference for the use of certain resources over others.

In the Prolog constraints defined above, when this type of reasoning is included, the definition of the `branchStrip` predicate is extended to refer to a predicate that pairs the summary resource requirement with each plan in the list of options. The list of plan options is sorted so that the subgoal branches nearest the leaves at the bottom of the tree are considered first. This is to reduce the number of plans being considered at each iteration through the list and to allow for simplified predicates summing the resource requirements as they do not need to consider branches at lower subgoals. Once the list of plan options paired with resource requirements is formed, it is then sorted into order of increasing resource requirements so the first element in the list is the preferred plan and the remaining plans can again be retracted.

```
branchStrip([]).
branchStrip([H|T]):-
    branchList(H,L),
    sort(L,[_|T2]),
    rmBranch(T2),
```

```
    branchStrip(T).

branchList([],T):-T=[].
branchList([P|T],T1):-
    branchList(T,T2),
    subtree(P,X),
    resAll(S,X),
    append([S/P],T2,T1).
```

The `resAll` predicate starts by producing a single long list of the resource requirements for each plan. For each plan, this takes the pairs representing the type of resource and quantity required and appends them to a list of all the resource requirements for the sub-tree being considered. Once all the resource requirements have been compiled into one list, this is sent to a summing predicate to simply add together all the quantities to produce a total resource requirement. It is in this final predicate where weightings could be included if necessary to indicate any preferences for which types of resources should be used.

```
resAll(S,Ps):-
    resourceList(L,Ps),
    resSum(S,L),!.

resourceList(L,[]):-L=[],!.
resourceList(L,[SG|T]):-     % Ignore subgoals
    node(SG,_),
    resourceList(L,T).
resourceList(L,[P|T]):-
    node(P,_,_,_,R),
    resourceList(L1,T),
    append(L1,R,L).

resSum(S,[]):- S=0,!.
resSum(S,[_/X|T]):-
    S#=X+S1,
```

```
resSum(S1,T).
```

After the plan options have been removed, the resource reasoning is next used to consider which goals can be safely adopted given the quantity of each resource available. The reasoning is performed in this order to firstly reduce the number of plans being considered and secondly to allow the summary information generated for reasoning about goal adoption to represent the actual requirements of the goal.

The list of top-level goals can be ordered in the same manner as the list of plan options for selecting the plans or, in this case, goals with the lowest resource requirements. To do this, the first step as before, is to generate the list of plans in the tree for each goal. This can be performed using the `branchList` predicate with a list of the top-level goals. This will pair up each of the goals with a number representing the sum of resource requirements regardless of type. It is possible to apply different orderings to the list of goals to indicate the importance of a goal, thereby preferring to complete less goals of greater importance than to achieve more goals of less importance. If the order in which the goal are considered for adopting is not important, or if the order is predefined by the order in which the goals were defined this step can be skipped. This will also provide a decrease in the number of steps and hence the length of time taken to evaluate the problem each time to find a solution. In the evaluation of this approach, the sorting and ordering was included in the reasoning.

The main reasoning about resources for goal adoption requires summary information broken down by the different types of resources required. This is so that the reasoning can check that there is actually sufficient resources available for each goal to be adopted. For each goal in the list, the summary information separating the different types of resource information is generated. While the `resAll` predicate produces a combined summary of each of the resource types into one number, the `resType` predicate used here, keeps the different types of resources separate when generating the summary information. The summary information produced by the predicate `resType` is an unsorted list containing each of the resource types and the quantity of it required by the goal, for example `S = [r3 / 6, r2 / 5, r1 / 7, r5 / 0, r4 / 0]`. From this list, each of the types of resource is extracted and compared to the quantity of that resource available.

```
resReason(G):-
    goalPlans(G,P),
    resType(S,P),                          % generate resource summary by type
    member(r1/A,S),                        % unify the resource values
    member(r2/B,S),
    ...
    resource(r1,RA), RA#>=A,               % check sufficient available
    resource(r2,RB), RB#>=B,
    ...                                    % reserve resources
    retract(resource(r1,RA)),  NewRA #= RA-A, asserta(resource(r1,NewRA)),
    retract(resource(r2,RB)),  NewRB #= RB-B, asserta(resource(r2,NewRB)),
    ...
```

If each type of resource has sufficient resources available then the predicate `resReason` will succeed and the quantity of each of the resources available will be lowered accordingly. If one or more types of resource has insufficient available then the predicate will fail and the *if-then-else* construct from which the predicate was queried (`resReason(G) -> true; strip(G)`), will step to the *else* component where the goal will be retracted in the same way as removing the sub-tree of a plan that is not required. After all the goals have been considered, adopting those that are safe to start, and removing those which are not, the reasoning then returns to the core part of the goal-plan tree representation to schedule the plans for the goals that have been adopted.

5.4 Modelling Positive Interaction Reasoning

The positive interaction attempts to identify plans in different goal-plan trees that can be 'merged' as they produce the same effects, as was described in section 3.3. When referring to plan merging, it is actually possible to achieve the effects by only using one of the two plans. By doing this, the number of plans required to achieve all the goals adopted can be significantly reduced, especially as the sub-trees of the plans that are not used are also removed when the two plans are merged. If the interaction between the goals occurs at a high level in the goal-plan trees, i.e.

near the root with each plan itself having a large sub-tree, then the impact of the merging is particularly significant.

To perform the reasoning in Prolog, a predicate is defined that identifies pairs of plans that produce the same effects by checking the lists of effects for the two plans are equivalent. This starts by unifying two plans and the list of effects generated by each of the plans, checking that the two plans are not the same plan. The reasoning cycle in Prolog when requested for all pairs of positively interacting plans will iteratively test every pair of plans including every plan with itself. This last test will fail on the constraint `Px\=Py`, where `Px` and `Py` are the names of two plans, and trigger a backtrack to the selection of the plan `Py` to try a different plan. For pairs of different plans, the effects of the plans are considered to identify if there is any possibility of merging them. Firstly, it is checked that the list of effects for the first plan is not empty, otherwise all plans that don't themselves achieve effects could be included for merging. Where an effect is produced by `Px`, the list of effects for the two plans are compared to see if they are equivalent. If so, then with all the constraints satisfied, the pair of plans is returned as a pair of positively interacting plans that can be merged. If the effects are not equivalent, then the solver backtracks again to try another pairing until all possible pairings have been tested.

```
pos(Px,Py):-
    node(Px,_,_,XEffects,_),
    node(Py,_,_,YEffects,_),
    Px\=Py,
    not(XEffects=[]),
    seteq(XEffects,YEffects).
```

The `findall([Px,Py], pos(Px,Py), Merge)` predicate is used to generate a list all the pairs of plans where it is possible for them to be merged. The template used to form the list from the solutions to the `pos(Px,Py)` predicate, places each solution pair of plans into its own sub-list. The complete list of positively interacting plans is then used to select and remove plans that are not needed as the effects they produce are duplicated by other plans. By default, the second plan in the pair of interacting plans is retracted, however this is not always the case.

While in the positive interaction reasoning considered here all the effects in the list must match for the plans to be considered for merging, it is also possible to consider a weaker version of positive interaction where only some of the effects match. In this case, in order to ensure that a plan that is kept from the merging with another plan is not then deleted by a later merging, the plan is "marked". This is done by asserting the predicate `mark(Plan)` for each of the plans that has been kept from a merged pair. When a pair is first considered, it is checked to see if either plan is already marked. If both plans are already marked, then neither plan can be safely removed as it is possible that the intersecting effect that was used to identify the two plans as positively interacting is different to the intersecting effects from the interactions where they have already been "merged".

As the reasoning here checks that the effects are equivalent, it is not necessary to check if one or both plans are already marked. This is because if one plan is marked, and has appeared in more than one positive interaction then the effects of three or more plans must all be equivalent, therefore only one plan is still needed to achieve the effects on behalf of all of the plans. However, as merges could occur within the sub-tree of one or both of the interacting plans, it is still necessary to mark the plan kept from a merge to ensure it does not get removed as part of a sub-tree.

The `posScheduler` predicate defined below starts by checking that the two plans both still exist and that one or both have not already been removed by other merges. The sub-trees of each plan are then generated to check for any marked plans within the sub-trees that could prevent one of the plans from being removed in a merge. If just one of the plan's sub-trees contains a marked plan, then that plan can be kept while the other is retracted, otherwise neither plan and its sub-tree can be removed.

```
posScheduler([]).
posScheduler([[P1,P2] | T ]):-
    node(P1,_,_,_,_), node(P2,_,_,_,_),
    subtree(P1,X), subtree(P2,Y),
    not((member(XP,X), mark(XP));
        (member(YP,Y), mark(YP))),
    ((not(member(XP,X), mark(XP)), asserta(mark(P1)), strip(P2));
    (not(member(YP,Y), mark(YP)), asserta(mark(P2)), strip(P1))),
    posScheduler(T).
```

When the reasoning about positive interactions is combined with that of reasoning about consumable resources, then the selection for which plan to keep and which plan to retract is influenced by the summary resource requirements for the sub-tree of each plan. In this case the predicate `resAll` is used to produce the summary information for the sub-tree of each of the two plans. The plan with the lower resource requirements is then kept when there is a free choice between the two plans as neither sub-tree contains any marked plans.

The positive interaction reasoning is incorporated into the set of constraints after the branch options have been removed. This is to reduce the number of matches as the branches provide different sets of plans for achieving the same effects within a goal-plan tree.

5.5 Modelling Negative Interference Reasoning

While the reasoning about positive interaction identifies plans that produce the same effects, the reasoning about negative interference identifies sets of three plans

where one plan generates the effect required by the second plan, and the third plan produces an opposite effect that if it were executed between the first two would cause interference. This can be thought of as a casual link between the first two plans, which the third plan would break, as described in section 3.4.

In Prolog, in order to identify the negative interactions between plans, the `neg(Px,Py,Pz)` predicate is defined to find pairs of plans that have causal links and the plans that can interfere with those links. `Px` is the plan that starts the causal link by producing the desired effect required as a precondition for plan `Py`. Once `Py` has executed, it is assumed that the effect is no longer required, so can be safely altered by other plans such as `Pz`. If however `Pz` attempts to execute between `Px` and `Py` then this will cause interference possibly leading to plan and then goal failure. As with the positive interaction reasoning, it is important to check that the plans are all different before comparing the preconditions and effects of the plans. To compare the effects, it is important to split up the pair notation for representing the effects of plans into the two component parts, the factor identifier and the value representing its current state (eg. `e1/7`). The `member(Element, List)` predicate, in the reasoning predicate shown below, unifies factors of the environment that are common to all three plans, but where the value assigned to that factor is different in the interfering plan to the value used by the linked plans.

```
neg(Px,Py,Pz):-
    node(Px,_,_,XEffects,_),
    node(Py,_,YPrecon,_,_),
    node(Pz,_,_,ZEffects,_),
    Px\=Py,Px\=Pz,Py\=Pz,
    member(V/N1,YPrecon),
    member(V/N1,XEffects),
    member(V/N2,ZEffects),
    N1#\=N2.
```

This predicate is again queried with the `findall([Px,Py,Pz],neg(Px,Py,Pz),Neg)` predicate to generate a list of all the possible instances of the interference so they can be scheduled to ensure the interference is avoided. For this, the interfering plan either needs to be scheduled to execute before the other plans or after both have

executed so the effect is no longer required. This is handled by the `negScheduler` predicate shown below.

```
negScheduler([]).
negScheduler([[Px,Py,Pz]|T]):-
    g_read(Px,A),
    g_read(Py,B),
    g_read(Pz,C),
    A#<#B,(C#<#A;C#>#B),
    g_assign(Px,A),
    g_assign(Py,B),
    g_assign(Pz,C),
    negScheduler(T).
```

The `negScheduler` predicate refers to the finite domain global variables that have been defined for representing the domain of values that can be assigned to each of the variables representing the plans for generating a schedule. The plan, `Px`, producing the effect must always occur before the plan, `Py`, using the effect. However, it is possible to schedule the interfering plan to either execute before `Px` or after `Py`, as long as it does not execute between the two plans.

The reasoning about negative interference is incorporated into the set of constraints after the tree scheduling has been performed. This is to ensure the minimum number of plans are considered as the evaluation of the `neg(Px,Py,Pz)` predicate considers all the possible combinations of three plans. In addition, the main purpose of the negative reasoning is to schedule potentially interfering plans to ensure they do not interfere, rather than reducing the number of plans, so this "scheduling" is performed after all the surplus plans have been removed and the schedule refined based on the constraints in the tree structure.

5.6 Constraint Automated Generation

For the purposes of evaluation, as with the Petri net model, it is necessary to automate the production of the instances of the constraint-based model in order

to be able to evaluate the different tree structures and settings considered in a reasonable length of time (see chapter 6). The structure of this model allows the automation to be simplified as the constraints are written in a plain text format where the main component to change is the goal-plan tree representation. By separating the different aspects of the constraint definitions into different files, the major changes for the goal-plan tree representation can be limited to one text file, with the fixed components remaining untouched in separate plain text files. The fixed components include the predicates for the different types of reasoning, and a set of utility predicates for set operations that had not been predefined in GNU Prolog. The file with the reasoning predicates will change depending on the types of reasoning selected for inclusion, but the predicates for each type of reasoning in them are fixed.

As most of the constraints remain the same, the most complex part of the automated generation for this model is the generation of the goal definitions themselves. Each plan and goal needs to have a unique ID so that it can be individually referenced, hence each node is prefixed with a two letter code to identify the top-level goal it belongs to, and a number to refer to the specific plan or subgoal within that goal. The tree is then traversed to list the subgoals and plan options for each of the plans and goals, along with the preconditions, effects and resource requirements for each of the plans. It should be clear that generating the list of goals and plans is done in one pass of the goal-plan tree making this linear with respect to the size of the tree.

Chapter 6

Evaluation

In this chapter we evaluate the performance of the two models when considering each of the types of reasoning separately and combined together. Presented in [Shaw and Bordini, 2008, Shaw *et al.*, 2008] are some preliminary results for the Petri net model, using both an abstract scenario and a more concrete Mars rover scenario. Due to the large number of new experimental results covering both models, the preliminary results from the papers are not included in this thesis. As with the results presented in this chapter, the three types of reasoning individually and combined in the Petri net model for the two scenarios show improvements in the performance when compared to the Petri net model without any reasoning, with only a small increase in time in some cases else a reduction in time taken. Summaries of all the results are given in section 6.5.

6.1 Experimental set-up

This section discusses the experimental set up that was used to test the two models described in chapters 4 and 5. The aim of this experimental analysis is to measure the performance of the models under highly constrained conditions as well as attempting to identify subclasses of the goal-plan tree problem where one model may be more suitable than the other. To this end, three different structures of goal-plan trees have been used to compare performance in different classes. The three tree structures are a deep tree, a broad tree and a general tree that is a cross

between the deep and branching trees to test the scalability of the two approaches.

The experiments cover each of the types of reasoning discussed in chapter 3 individually as well as each of the possible combinations of them working together. The data recorded from the run time results was the number of goals successfully achieved, the number of plans used, the time taken and the sizes of the generated files. Also recorded were the memory and processor usage and the time taken to load the files in order to run each of the models.

When measuring processor usage for the two models it was noted that despite the computers used to run the experiments having dual core processors (specification given below), the applications (i.e., Renew for simulating the Petri net models and GNU Prolog for evaluating constraint models), were only able to make effective use of a single processor at a time. The processor they did use was used to the maximum in both models for the duration of the run time after which the usage dropped back down to 0.

The memory requirements recorded took into account the standby memory for when the files were loaded before the experiments were run, and the runtime memory usage whilst the simulations were running. The basic memory requirements for each application were recorded without any files loaded. This was 36.88 Mb for the Petri net editor Renew, and 4.89 Mb for the Prolog editor GNU Prolog.

As was stated in section 4.6 of chapter 4, the loading of the Petri net model takes two steps, first importing the three PNML files then saving them to the Renew file format with specific names to allow the cross referencing between the three Petri nets. The timings of these operations are discussed along with those for the constraint-based model when comparing the two approaches, however the graphs presenting the timings only show the run times for performing each of the experiments. The break down of the load times for each model are presented in tables accompanying the graphs, with the graphs showing the runtime measurements of time taken, goals started & achieved, and plans used.

To evaluate the effectiveness of the different types of reasoning under different conditions there are a large number of parameters that can be varied. These include tree size, reasoning type, resource availability, height of positive interaction, duration of negative interference, amount of interaction between goals and finally the number of goals. Due to time constraints, it was not possible to evaluate

all possible combinations of parameters so a subset has been selected to analyse performance where the greatest effectiveness of the reasoning for each given tree was expected to be. The combinations of parameters that were considered and the experiments that were performed are detailed in tables 6.1–6.4, with the results for each given in the sections 6.2–6.4.

Where appropriate, some statistical analysis of the results is performed. This is based on the Coefficient of Variation (CV) that provides a statistical measure of the dispersion of data points in a data series around the mean and is often expressed as a percentage [Anderson *et al.*, 2007, Chapter 3]. This allows the comparison of different variables with different standard deviations and means. The formula for calculating the CV is shown in equation 6.1 where σ is the standard deviation of the data series and μ is the mean. This is multiplied by 100 to give the percentage CV.

$$\frac{\sigma}{\mu} \times 100 \tag{6.1}$$

This formula works well with high mean values that show the percentage of variance from this mean, such as for analysing plans used and goals achieved. However, when the mean is less than the standard deviation for a set of results, the resulting percentage is greater than 100 and provides very little meaningful interpretation of the results. In these cases, a separate analysis is used to provide a more meaningful interpretation of the data based on the range of values in the data set. For example, in an experiment for reasoning about resources (see section 6.2.1), the random Petri net model achieved an average of 0.13 goals, with a standard deviation of 0.35. This gives a CV of 263.9%, when a more meaningful analysis is simply that the range of goals achieved is between 0 and 1.

Tree Size Within the deep and broad trees, the tree sizes are varied in some of the experiments, with the medium tree size being the main size used in most of the experiments. This is to evaluate how the performance of the reasoning varies depending on the tree size. The small tree sizes have approximately 25 plans, while the medium trees have approximately 50 plans and the large trees have approximately 100 plans. The final tree structure is only used as a large tree,

with 94 plans, to further evaluate how well the two models for reasoning scale, particularly when adding in more goals.

Goal Interaction Level The level of interaction between goals was varied through the use of a set of variables. Where positive and negative reasoning were being evaluated the effects on the environment was simulated through the use of variables representing the attributes in the environment that were being changed. There are 5 variables that represent the attributes that can be changed within the environment. At low levels of goal interaction, each goal only modifies 1 of these variables, while at medium level each goal modifies 2 variables and at high level each goal modifies 3 variables. This means that, when there are 20 goals, at low levels of goal interaction each variable is being modified by 4 goals, at medium level this increases to 8 goals modifying the same variable and at high levels there are 12 goals competing over the same variables. This results in greater interaction between the goals as more goals are attempting to access and modify the same variables. The distribution of goal interactions on the variables was kept at an even level, such that each variable was accessed by the same number of goals.

Resource availability The amount of resources available was varied to analyse how the reasoning was able to perform under highly constrained conditions, and the cost of the reasoning when the availability is high. This was set to three levels; low, medium and high availability. The low level was set to approximately 30% of the total resource requirements, while the medium level provided approximately half the required resources and the high level provided approximately 85%. In a similar way to the goal level interaction, there were 5 different types of consumable resources available, each starting with the same quantity of resources. The number of goals consuming each type of resource was varied in the same way as the goal interaction level to simulate the interaction when only resource reasoning was being evaluated. The amount of each resource being consumed by a given plan was varied between 1 and 5 units such that the goals had approximately the same overall resource requirements but they may require more of one type of resource than another.

Positive Interaction Level This was simulated by setting different goals to assign the same values to variables, thereby achieving the same effect on the environment. The height within the tree at which this interaction occurred is used to vary the overall impact that this interaction has on the number of plans used. If the interaction occurs near the root at the top of the goal-plan tree, the effect is more dramatic as a greater number of plans can be dropped from the sub-trees of the interacting plans, significantly reducing the total number of plans used. The three levels used here are high, middle and low level, where high level refers to interactions occurring near the root, around levels 3 and 4 in the deep tree (see figure 6.1), and level 2 in the broad and general trees (see figures 6.12 and 6.18). At low level, the interaction occurs at the lower levels of the tree, around level 14 in the deep tree, and levels 4 or 5 in the broad tree and general tree respectively. The mid-level interaction is part way between the two extreme levels. In the general and broad tree in particular, there is very little variation in the level due to the relatively shallow nature of the trees, so the changes to this variable are most visible in the deep tree.

Negative Interference Duration As with the positive interaction, this is simulated by setting different goals to assign values to the variables. In this case however, the values assigned by each goal are different, and the goal must read the same variable at the end. If the value has changed between writing and reading it then the plan fails causing the goal to fail as well. To obtain the longest duration, the reading of the variables is done at the lowest level in the tree, while the level at which the variable is written is changed in the same way as was done in the positive interaction. In the low level interaction, this is changed to writing the variables at the penultimate level, rather than the last level in the tree where the variables are now being read, or by moving up the point at which the variables are read. The fewer plans or the shorter the duration between the writing then reading of a variable, the greater the probability of that goal being achieved successfully without any reasoning. As with the positive interaction, the effects on the duration from changing this variable are most visible in the deep tree. It should be noted here that the plans themselves are not given any execution time, the duration simply arises from the distance between the level in the tree at which

		Consumable Resource Reasoning								
		Availability			Interaction			No. Goals		
		High	Med.	Low	High	Med.	Low	High	Med.	Low
Deep Tree	Small									
	Medium	✔	✔	✔	✔			✔	✔	✔
	Large									
Broad Tree	Small			✔	✔				✔	
	Medium			✔	✔				✔	
	Large			✔	✔				✔	
General Tree	Small									
	Medium									
	Large	✔	✔	✔	✔				✔	

Table 6.1: Settings considered in experiments for reasoning about resources

the variable is written and the level at which the variable is read. The greater the difference between the levels, the longer the duration that the variable would need to be protected.

The Petri net model experiments were each repeated 15 times to allow for the slight variation between repeats, averaging them to avoid distortion, while the experiments for the constraints model were only repeated 3 times each. This was due to the consistency of the results returned each time, generating the same solutions to achieving the same number of goals using the same plans. The only variation came in the duration of the reasoning which was averaged out over the 3 repeats. In terms of the total duration of the executions however, this variation was very small.

The variation in the number of plans or goals achieved by the Petri net model is caused by a certain amount of random selection. Where there are choices between two equally good plans at a subgoal within a goal the model randomly selects between the two, while the constraints model always selects the first available. Also, the order in which the goals are started is not fixed in the Petri net model, even in the scenarios involving limited consumable resources. This means that the Petri net model could start goals with large resource requirements first, or take on several goals that all require a lot of the same resource, rather than starting goals with a more evenly distributed demand for resources. In future work, this

		Positive Interaction Reasoning								
		Position			Interaction			No. Goals		
		High	Med.	Low	High	Med.	Low	High	Med.	Low
Deep Tree	Small	✔			✔				✔	
	Medium	✔	✔	✔	✔				✔	
	Large	✔			✔				✔	
Broad Tree	Small									
	Medium	✔			✔			✔	✔	✔
	Large									
General Tree	Small									
	Medium									
	Large	✔			✔				✔	

Table 6.2: Settings considered in experiments for reasoning about positive interaction

		Negative Interference Reasoning								
		Duration			Interaction			No. Goals		
		Long	Med.	Short	High	Med.	Low	High	Med.	Low
Deep Tree	Small	✔			✔				✔	
	Medium	✔	✔	✔	✔	✔	✔		✔	
	Large	✔			✔				✔	
Broad Tree	Small									
	Medium	✔			✔			✔	✔	✔
	Large									
General Tree	Small									
	Medium									
	Large	✔			✔				✔	

Table 6.3: Settings considered in experiments for reasoning about negative interference

		Resource			Positive			Negative								
		Availability			Position			Duration			Interaction			No. Goals		
		High	Med.	Low	High	Med.	Low	Long	Med.	Short	High	Med.	Low	High	Med.	Low
Deep Tree	Small															
	Medium			✔	✔			✔			✔				✔	
	Large															
Broad Tree	Small															
	Medium			✔	✔			✔			✔				✔	
	Large															
General Tree	Small															
	Medium															
	Large	✔	✔	✔	✔			✔			✔			✔	✔	

Table 6.4: Settings considered in experiments for combined reasoning

is something that could be modified to give some kind of ordering to the goals, either by adding a weighting indicating importance of a given goal or controlling the order in which goals were started to optimise the number of goals achieved. Part of this difference comes from the different styles of the approaches used to model the reasoning. The CSPs are essentially sequential, selecting the first valid assignments, while Petri nets are essentially concurrent.

Where timings are taken, the timing does not include any time for the actual execution of the plans, purely the time taken for the Petri net simulation or constraint evaluation to complete. The hardware used was a set of six machines, each with 2.66 GHz dual core processors and 3.5GB RAM, running Linux Ubuntu 8.04 (Hardy) and Gnome Desktop 2.22.2. GNU Prolog version 1.3.1 and Renew version 2.1 were used for running the models themselves.

The aim of these experiments is to stress test the different types of reasoning in the two models under different conditions to analyse their performance. The test results are presented and evaluated for each of the different tree structures in turn before comparing the performance between the different tree structures. This starts with a thorough investigation using a deep tree to illustrate the effect of the reasoning over the absence of reasoning in the Petri net model. This is then followed by a comparison using a broad tree as opposed to a deep tree structure for the two models. The deep tree provides a large number of sub-plans and

subgoals, with very few choice branches, while the broad tree offers a lot of choices between different branches in the tree while having very little depth. The final tree structure used in these experiments is a cross between the straight depth of the deep tree and the large scale branching of the broad tree. This tree is mainly used to analyse how well the reasoning scales in general by using a large tree structure and a large number of goals.

Within each tree structure, the three types of reasoning are each considered individually, before looking at the effects produced by combining two or more of the reasoning types. An attempt has been made to select the most relevant experiments out of the thousands of possible variations of settings, to show where the largest potential gains can be made from applying the reasoning. Therefore, selection is based on the tree structure and the type of reasoning being considered.

6.2 Deep Goal-Plan Trees

The deep tree model used here is aimed at evaluating the performance of the two approaches where the depth of plan paths required to achieve each individual goal ranges from a depth of 6 for the small tree size to 15 for the large tree size, as illustrated in figure 6.1. At any level across the breadth of the tree, the number of branching plans or subgoals is at most two, with most levels containing either 0 or 1 branches and where there is a branch, there are just two options to choose between. This allows for a certain amount of choice within the tree, without the tree getting too broad so that the main attention is on the depth. Most of the branching also occurs at the plans so both subgoals have to be achieved rather than at the subgoals giving an option of plans to use. This means that there is very little variation in the minimum and maximum number of plans required to achieve each goal individually, as shown in table 6.5.

The main effects illustrated from the use of this tree structure are related to negative interference spread over a long duration, stretching from a high level in the tree down to the leaves at the bottom of the tree. Reasoning singly about consumable resources is unlikely to make any difference to the number of plans used to achieve each goal due to the small number of places where this reasoning can be used within the tree structure. However when combined with reasoning

Size	Depth	Total Plans	Total Subgoals	Min. Plans req.	Max. Plans Req.
Small	6	23	21	22	22
Medium	10	50	47	38	41
Large	15	96	90	62	66

Table 6.5: Plan requirements for the three sizes of deep tree used

over positive interactions at high levels in the goal-plan tree there is likely to be a much greater impact for the number of goals achieved when applying the resource reasoning over the limited availability of the consumable resources.

The remainder of this chapter is dedicated to presenting and analysing the results generated from the experiments described above. The results are all presented uniformly to aid reading, with a set of graphs for the Petri net model followed by those for the constraint model. Each set contains the graphs for running times, goals started & achieved, and the number of plans used. For ease of reference, a consistent colour scheme for each of the graphs has been used, shown in figure 6.2. This is then followed by two tables, the first showing the loading times for each model, while the second shows the memory usage.

6.2.1 Consumable Resources

In this section we focus on the application of reasoning about resources in the two models developed in chapters 4 and 5. The effectiveness of this type of reasoning within the deep tree is measured by first varying the number of goals vying for the same resources, before varying the availability of the resources for a fixed number of goals.

Varying Number of Goals

The first set of experiments involves the reasoning about the limited availability of consumable resources. In this set of experiments the number of goals was varied between 10, 20 and 30. The medium tree size was used with high levels of interactions between the actual goals, and low availability of resources. The results for the Petri net and constraint models are shown in figure 6.3.

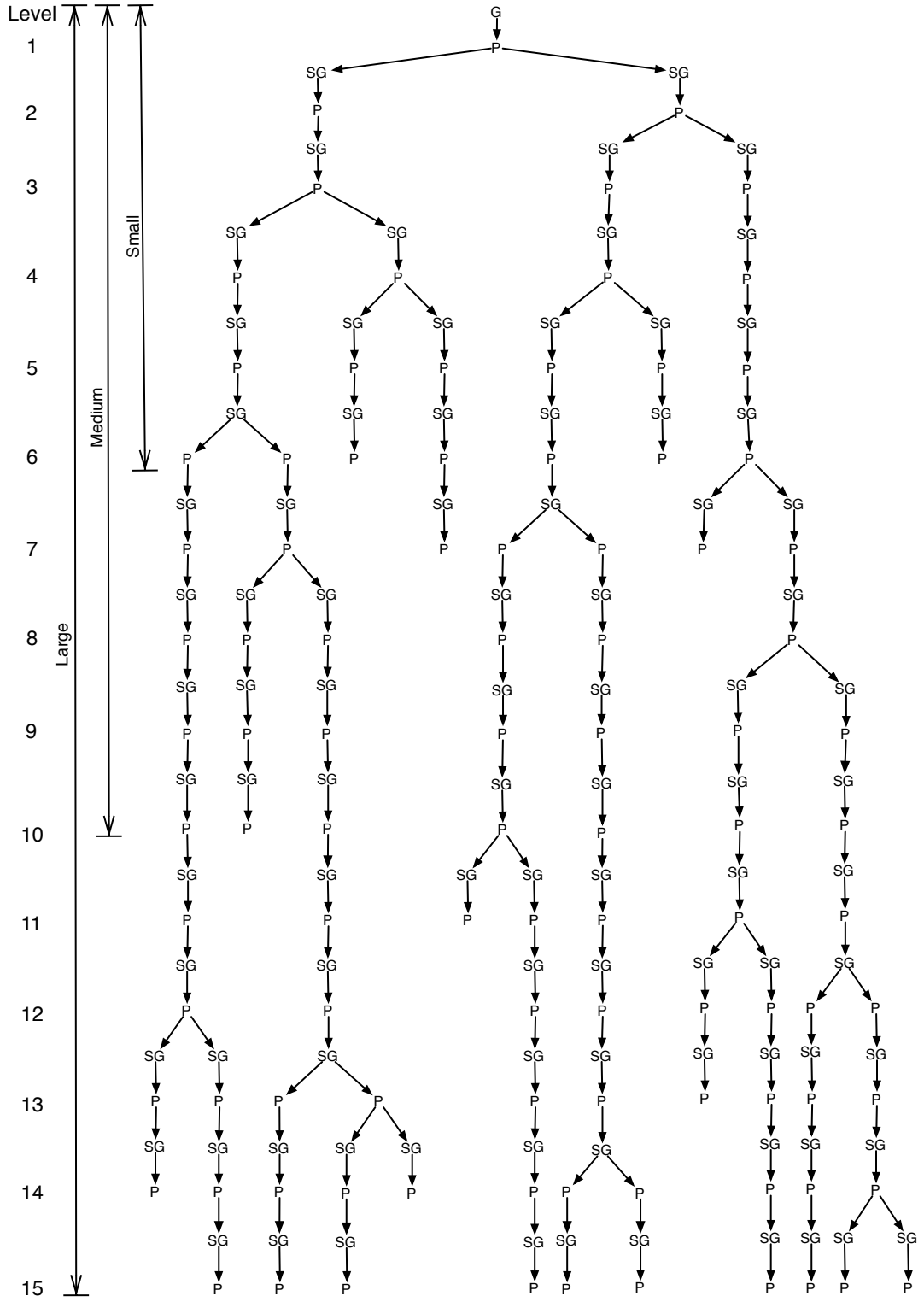
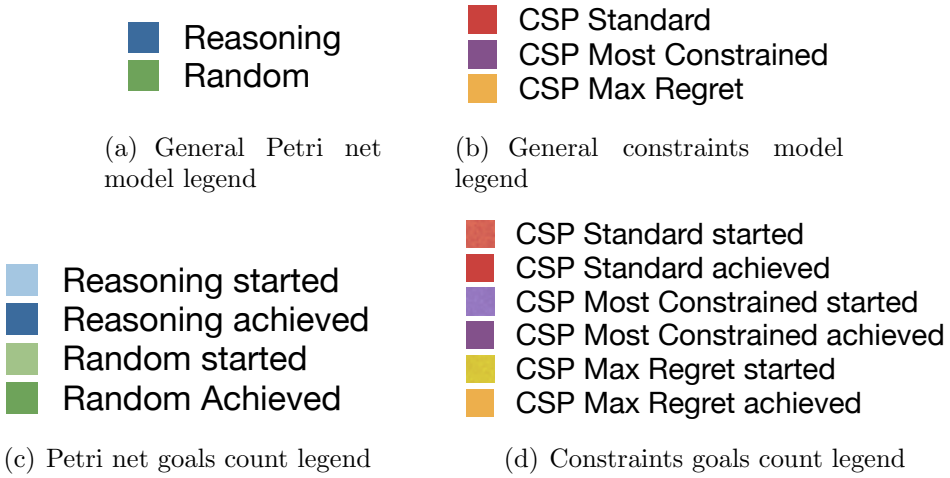


Figure 6.1: Deep goal-plan tree showing the levels used for small, medium and large goals



(a) General Petri net model legend

(b) General constraints model legend

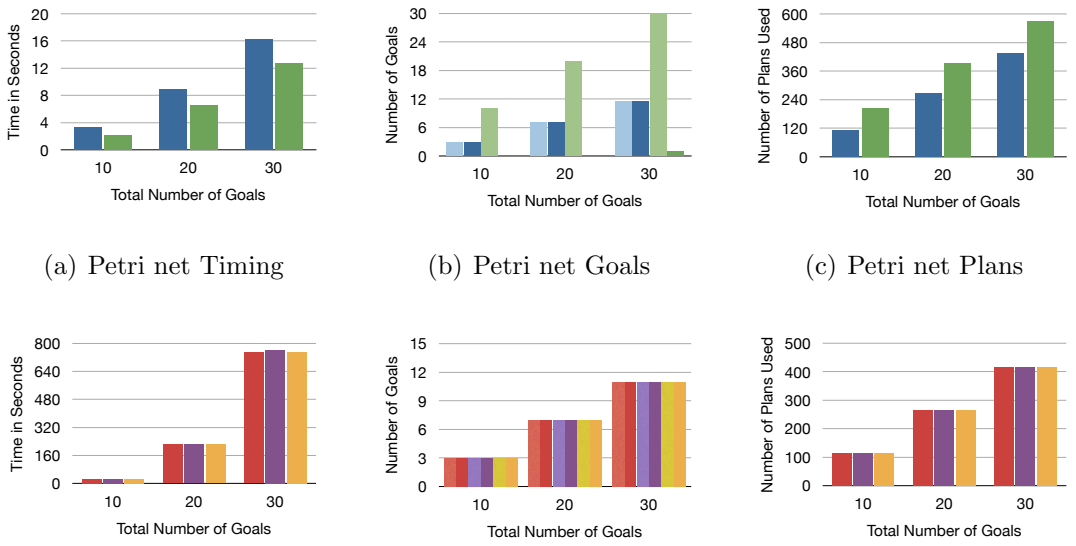
(c) Petri net goals count legend

(d) Constraints goals count legend

Figure 6.2: The different legends for the result graphs of the Petri net and constraint models

The first point to notice from the results shown in figure 6.3(b) is the difference between goals started and goals achieved for the reasoning and random Petri nets. The reasoning Petri net only started goals it was able to achieve given the limited availability of resources, while the random Petri net attempted to start all goals. As a result, very few goals were actually achieved by the random Petri net, achieving just 1 goal out of the 20 started in two of the repeats, and an average of 1 goal over all out of the 30 goals started. In nearly half of the repeats from the Petri net experiments with 30 goals, the random model did not achieve any goals, only achieving 1 goal in a quarter of the repeats. In two of the repeats however, 4 and 5 goals were achieved indicating that is possible for a random selection to achieve some goals occasionally. The results for the reasoning Petri net were much more consistent, with all but one repeat achieving 3 goals out of 10, the other repeat achieving 2 goals, similarly all but one repeat achieving 7 goals out of 20, with the other repeat achieving 8. Out of the repeats for the experiments involving 30 goals, approximately half the repeats achieved 11 out of 30 and the other half 12 out of 30 for the reasoning Petri net. This is compared to the 11 goals out of 30 achieved by the constraint-based model.

As the total amount of resources increased, while still being limited proportionally to the number of goals set, there was a greater chance for a goal adopted



(a) Petri net Timing (b) Petri net Goals (c) Petri net Plans

(d) Constraint Timing (e) Constraint Goals (f) Constraint Plans

Figure 6.3: Results for setting: Medium sized deep tree, low resource availability, high goal interaction, varying number of goals and reasoning about resources

early to be successful in the random Petri net resulting in the higher success rate as the number of goals increased. These two occurrences were probably also made possible by the large number of plans available for selection resulting in the random occurrences where the plans selected did not compete for the same resources allowing some goals to be occasionally achieved. As the average shows, this is the exception and not the norm. Compared to the total number of goals started though, and the total number achieved by the reasoning Petri net and constraints model this is still not an acceptable success rate for the purely random selection approach.

Comparing the results of the reasoning Petri net with those of the constraint-based model show much greater consistency in the number of goals achieved. The constraint model consistently achieved 11 goals out of 30 using 418 plans, while the reasoning Petri net achieved an average of 11.5 goals using an average of 436 plans, or more precisely, 418 plans for each repeat where 11 out of 30 goals were achieved, and exactly 456 plans where 12 out of 30 goals were achieved. This equates to 38 plans per goal and is in fact the minimum number of plans required

to achieve the medium sized deep tree goal (see table 6.5 for the table of plans per goal for the deep tree). The same is also true for the experiments using 10 and 20 goals, with both reasoning Petri net and constraint model using 114 plans to achieve 3 out of 10 goals, and 266 plans to achieve 7 out of 20 goals.

When considering the timings for the running of the experiments, the reasoning Petri net does add an increasing amount of time on to the time taken by the random Petri net, 1.2, 2.4 and 3.6 seconds respectively. Along with the time actually taken for the reasoning itself, the main reason for this should be clear as the extra goals achieved will cause an increase in the time taken. Conversely, when comparing the number of plans used between the reasoning and random Petri nets the explanation is less clear as the random Petri net uses a greater number of plans yet achieves a considerably smaller number of goals. The reason though is quite simply that plans are being executed in an attempt to achieve all the goals in the random model and not all the plans will consume any resources. Looking at the plan to goal ratio though, with an average of 20 plans per goal adopted across all three sets, shows that the average number of plans executed per goal adopted is insufficient for the goals to have been achieved.

As is shown in figure 6.3(d), the effect of the different heuristics used by the constraint solver was minimal. They made no different to the number of plans used or goals achieved, and very little difference to the time taken. This difference is most noticeable in the longer runs. In the experiment where there are 30 goals, the *Most Constrained* heuristic took slightly longer than the other two. To be precise, it was 12 seconds slower than the standard heuristic and 10 seconds slower than the *Maximum Regret* heuristic. This is just 1.5% of the total time taken, so is a relatively small difference. The CV values for the timings of each of the heuristics is less than 0.2% showing the very small variance between the three repeats using the constraint model.

The major difference however, occurs when the run times of the two different approaches are compared. The reasoning Petri net takes 3.04 seconds (CV 13%) to finish reasoning about 10 goals, while the constraint model takes 27 seconds (CV 0.2%). This increased to 9.05 (CV 10%) and 222 seconds (CV 0.1%) for 20 goals, and 16 seconds (CV 5%) in the Petri net model compared to 755 seconds (CV 0.15%) in the constraint model for 30 goals respectively. The coefficient of

variance decreasing in the Petri net model as the mean of the values increases. However, when including the loading times of the two models the comparison is much closer. As described in section 4.6, the production of the Petri net model is a two stage process, firstly generating the files using a Petri net interchange format that gives a concise definition of the Petri nets then importing these files into Renew. As the reference nets need the file names of the referenced nets to be specified the imported nets need to be saved using the file names specified in the manager Petri net. Together this means there are two additional timings for the Petri net model to import then save the files before they can be run. In the constraints model, the files generated are already in the appropriate format for GNU Prolog so can be loaded in one step ready for evaluation.

As can be seen in table 6.6, the time taken to import and save the files for the Petri net model is considerably longer than that taken by the constraint model. The total time taken to import and save the files is actually longer than the time taken for the constraint model to find a solution, however the difference in the time taken between loading the Petri net and the constraint solver reduces as the number of goals increases. Part of the difference in loading times comes from the file sizes produced for each model. The Prolog files for 10 goals are just 32Kb, increasing to 87Kb for 30 goals, while the PNML files that are imported to Renew start at 699Kb for 10 goals and increase in size to 2164Kb for 30 goals. Once these are imported into Renew and saved in the Renew file format, the file size increases even further up to 4723Kb for 10 goals and 9426Kb for 30 goals.

	Petri net model (seconds)				Constraints model (seconds)		
	Reasoning		Random		Standard	Most	Max.
	Import	Save	Import	Save		Constr.	Regret
10	22	26	21	26	0.116	0.112	0.114
20	88	212	87	207	0.307	0.306	0.310
30	222	686	218	531	0.586	0.603	0.601

Table 6.6: Load timings for setting: Medium sized deep tree, low resource availability, high goal interaction, varying number of goals and reasoning about resources

Comparing the time taken between the reasoning Petri net and the random Petri net models shows that the reasoning model does take slightly longer to load

with the additional transitions and places for the reasoning, the difference increasing slightly as the number of goals increases. Once the Petri net model has been loaded the time taken to run the experiments is much faster, allowing repeats to be performed in a reasonable length of time if desired. Once loaded the Petri nets can also be modified although currently this is a manual process that could potentially be automated in the future.

The files used by the constraint model are all the same size, regardless of which heuristic is being used. As a result the loading times for each heuristic only vary slightly due to variances caused by the operating system and application whilst loading the files.

	Petri net model (Mb)				Constraints model (Mb)					
	Reasoning		Random		Standard		Most Constr.		Max. Regret	
	Ready	Run	Ready	Run	Ready	Run	Ready	Run	Ready	Run
10	61.80	90.10	61.40	95.45	5.74	6.73	5.75	6.74	5.75	6.74
20	77.74	127.38	105.32	125.66	6.13	7.93	6.13	7.94	6.14	7.95
30	132.17	159.93	173.00	180.00	6.45	9.00	6.45	9.06	6.45	9.03

Table 6.7: Memory usage for setting: Medium sized deep tree, low resource availability, high goal interaction, varying number of goals and reasoning about resources

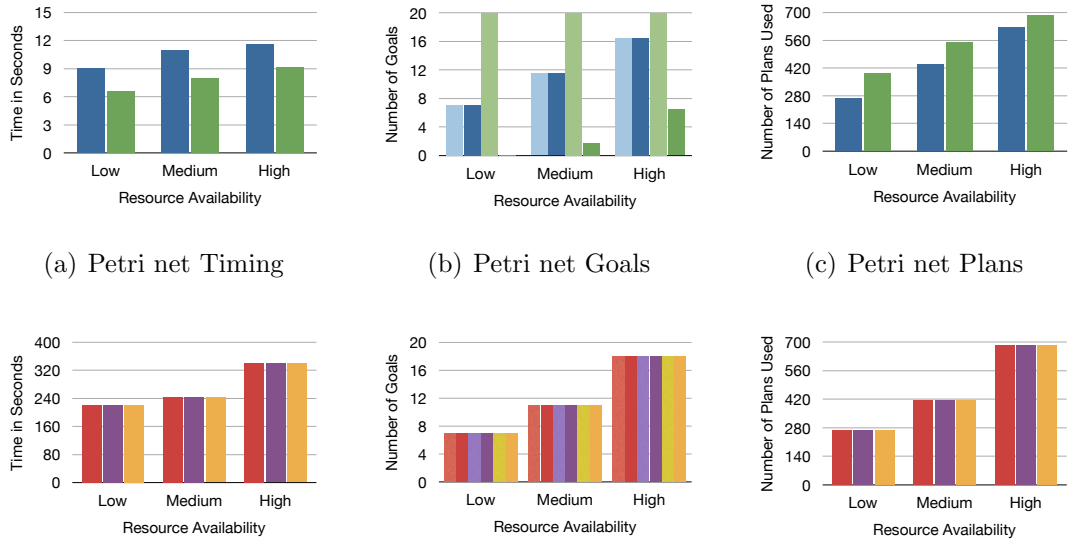
Similarly, the memory requirements for the constraint model have little variation once loaded and ready to run. When evaluating the constraints, the Most Constrained and Maximum Regret heuristics do require slightly more memory for the labelling of variables compared to the Standard heuristic, especially as the number of goals increases and the list of variables to assign values to grows in size.

In addition to the extra loading time for the Petri net model, the memory requirements for this model are considerably greater than those required by GNU Prolog to perform the constraint reasoning, as shown in table 6.7. Recalling the memory requirements stated above of the applications themselves, GNU Prolog requires approximately 4.89Mb of memory and Renew 36.88 Mb. Using this, the results show a small additional memory cost for the constraints model, while the increase for the memory usage in the Petri net model is more significant with an increase of 53.22 Mb when running 10 goals, increasing to 136.12 Mb for 30 goals. The trade off here is between the speed of the reasoning and the amount of memory

required. However, when comparing the memory requirements for the reasoning Petri net model to random Petri net there is a general decrease in the amount of memory required when the reasoning is included, especially when there are 30 goals, with a decrease of 41Mb when in standby and 21Mb of memory required to run the models.

Varying Resource Availability

In this second set, the number of goals in the system was maintained at 20, however, the availability of the resources was gradually increased to the point where there were sufficient resources available to achieve more than three quarters of the goals in the system, provided they were not wasted. The results for the Petri net and constraint model are shown in figure 6.4.



(a) Petri net Timing (b) Petri net Goals (c) Petri net Plans

(d) Constraint Timing (e) Constraint Goals (f) Constraint Plans

Figure 6.4: Results for setting: Medium sized deep tree, high goal interaction, 20 goals, varying resource availability and reasoning about resources

As before, the reasoning Petri net is only starting the goals that it is able to achieve given the resources available, and the number of goals started increases at the availability of the resources increases. This time, the random Petri net is able

to achieve more goals than in the previous experiments, ranging from 0-3 goals at medium availability and 3-9 goals at high levels of resource availability. However even at high resource availability, the random Petri net still achieved less than half the number of goals achieved by its reasoning counterpart. This is due to it still starting goals where there were insufficient resources available to complete them, thereby wasting resources and reducing the number of goals it can achieve. The difference in plans used between the two different Petri nets did decrease as the availability increased, showing that the random Petri net was getting closer to achieving all its goals with an average plan to goal ratio of 34 when given a high availability of resources. The reasoning Petri net was again using exactly 38 plans per goal achieved.

When comparing the number of goals achieved between the two approaches the results are the same except for the high availability, where the constraint model consistently achieved 18 goals while the reasoning Petri net model achieved an average of 16.5 goals, varying evenly between 16 and 17 goals on different runs. While the Petri net model does contain reasoning to stop it from starting goals it cannot achieve, the goals it selects to start are selected randomly while there are sufficient resources available to definitely be able to achieve all the goals started. This means that it is possible it will select goals with higher resource requirements or goals with similar resource requirements such that in the case of these experiments it runs out of one type of resource first, rather than attempting to optimise the selection. While the constraint model also does not necessarily optimise the number of goals achieved, it does sort them into order based on the sum of all the resources that each goal will need, and starts from the lowest requirements working up. This means that it may be able to achieve a few extra goals by accepting those with slightly lower resource requirements than the other goals, and leaving out those with the larger more expensive resource requirements.

An extension of this could be to consider the importance of each goal. Sorting the top-level goals based on how important they are to ensure that they are achieved first, before less important goals are executed and consume limited resources.

Comparing the running and loading timings between the two approaches again yields the largest difference with the reasoning Petri net taking 11.7 seconds (CV

4.7%) to simulate the model when resource availability is high, compared to the 340 seconds (CV 0.1%) required by the constraint model. There is a large jump in the timings for the constraint model between the medium and high availability due to the extra number of goals being achieved and the additional plans related to them. Between low and medium, there are just 4 extra goals, however between medium and high there are 7 extra goals, and this will add to the overall size of the computation being performed.

	Petri net model (seconds)				Constraints model (seconds)		
	Reasoning		Random		Standard	Most	Max.
	Import	Save	Import	Save		Constr.	Regret
Low	88	212	87	207	0.307	0.306	0.310
Med.	88	212	86	214	0.309	0.317	0.321
High	88	212	85	213	0.315	0.317	0.313

Table 6.8: Load timings for setting: Medium sized deep tree, high goal interaction, 20 goals, varying resource availability and reasoning about resources

Table 6.8 shows the file loading times for the two models, again showing a significant difference in the time taken to load the Petri net model files compared to the constraint model. The changes to resource availability has no effect on the load times or on the file sizes of either model, the file sizes being 1439 Kb for the PNML and 9425 Kb for the Renew file format, compared to 60 Kb for the GNU Prolog files. Comparing the total times taken for loading and reasoning, the loading portion, which takes the longest for the Petri net model, is constant regardless of the availability, while the constraint reasoning time increases as the availability and hence number of goals increases. As a result, the total time taken by the Petri net model at high resource availability is 311.7 seconds compared to 342.0 seconds in the constraint model.

Comparing the memory requirements of the two models, shown in table 6.9, again shows that with the exception of the Petri net reasoning model for low availability, which is an outlier, the standby requirements for both models are consistent. When simulating the Petri net model the memory requirements still remain relatively consistent despite the variation in the number of goals started. However, in the constraint model, the memory requirement again increases as the availability of resources increases and therefore the number of goals. This increase

	Petri net model (Mb)				Constraints model (Mb)					
	Reasoning		Random		Standard		Most Constr.		Max. Regret	
	Ready	Run	Ready	Run	Ready	Run	Ready	Run	Ready	Run
Low	77.74	127.38	105.32	125.66	6.13	7.93	6.13	7.94	6.14	7.95
Med.	105.55	141.18	105.24	124.80	6.14	8.03	6.14	8.05	6.14	8.05
High	106.31	126.88	105.00	126.53	6.14	8.20	6.12	8.23	6.13	8.24

Table 6.9: Memory usage for setting: Medium sized deep tree, high goal interaction, 20 goals, varying resource availability and reasoning about resources

still gives a memory requirement significantly lower than that used by the Petri net model.

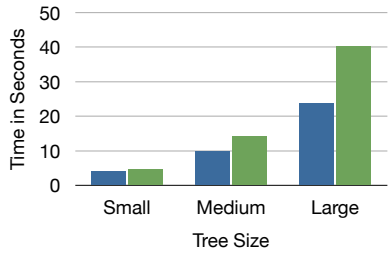
6.2.2 Positive Interaction

In this section, the focus is on reasoning about positive interaction. When reasoning about positive interaction, the experiments are set up such that all goals are achievable by both models, and the random Petri net. The key result being measured here is the number of plans used, and the effect this has on the time taken. As a result, the graphs showing goals started and achieved are omitted as all goals are always achieved.

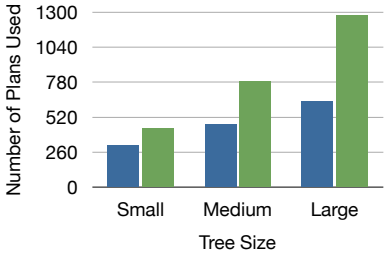
Varying Tree Size

The first set of experiments applying the reasoning about positive interaction focuses on the effects of varying the size of tree used by the goals. The different sizes of the deep tree can be found in figure 6.1 along with the plan requirements in table 6.5. The level within the tree at which the plan interaction occurs in these experiments has been set to a high level. This means that plans near the root of the goal-plan tree will interact. This causes the greatest effect as the size of the sub-tree from the plan that is dropped due to the merging is at its largest. If the interaction occurred near the leaves, this would make very little difference to the number of plans executed as there would be very few, if any, subgoals and sub-plans to drop. The results for the Petri net and constraint models are shown in figure 6.5.

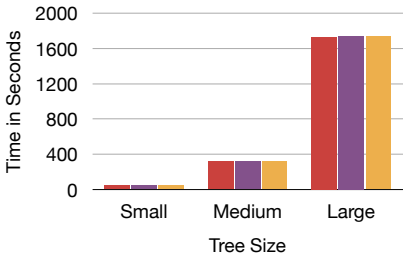
The results show that where the tree size is small the effect of the positive



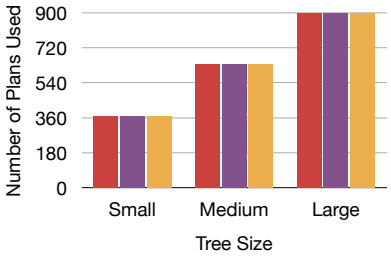
(a) Petri net Timing



(b) Petri net Plans



(c) Constraint Timing



(d) Constraint Plans

Figure 6.5: Results for setting: Deep tree, high level positive interaction, high goal interaction, 20 goals, varying tree size and reasoning about positive interaction

interaction is also small in terms of both plans saved and time taken. This is because the size of the sub-tree and hence the number of sub-plans beneath the interacting plans is quite small, so the saving is minimal. However, this saving increases rapidly as the size of the tree, and hence the sub-trees being dropped, increases. This is emphasised in the large tree where the number of plans used by the reasoning Petri net is almost half that of those used by the random Petri net. This saving is also echoed in the time taken. When fewer plans are being used, the length of time taken is also reduced, by as much as 16.4 seconds for the large size tree in the Petri net model.

In the small tree size, the random Petri net consistently uses 441 plans as all the possible branches in the small deep tree contain the same number of plans. The reasoning Petri net reduces this down to an average of 315.5 plans with a CV of 0.7%. This increases to an average saving of 318 plans for 20 goals with a CV of

2.9% and a saving of 636 plans in the large tree with a CV of 3.6%. The variance increases as the tree size increases due to the different sizes of the branch options resulting in more variance in the number of plans used.

Comparing the number of plans used between the reasoning Petri net model and the constraint model, it is clear that the constraint model has not achieved such great savings as the reasoning Petri net model has managed to obtain, instead requiring 900 plans for the large tree, 640 for the medium and 371 for the small tree. However, there is still a significant reduction in the number of plans used compared to those required by the random Petri net, with a 29.4% reduction in the number of plans used for the large tree, along with 15.8% and 18.9% for the small and medium sized trees respectively.

Compared to the reasoning about resources, both models take significantly longer to finish. This is mainly due to the larger number of simultaneous goals being considered and the number of plans involved as no goals are prevented from starting due to a lack of resources. This is most noticeable in the large tree size where the constraint model took 1731 seconds using the standard heuristic, and an additional 7-9 seconds for the other two heuristics. This equates to nearly half an hour for the solver to find a solution. Once loaded, the Petri net model only takes 24 seconds, however as shown in table 6.10 the importing and saving times for the Petri net model are again significantly larger than those for the constraint model.

	Petri net model (seconds)				Constraints model (seconds)		
	Reasoning		Random		Standard	Most	Max.
	Import	Save	Import	Save		Constr.	Regret
Small	24	57	20	21	0.110	0.109	0.108
Med.	93	217	80	191	0.318	0.313	0.313
Large	376	738	345	692	0.882	0.884	0.881

Table 6.10: Load timings for setting: Deep tree, high level positive interaction, high goal interaction, 20 goals, varying tree size and reasoning about positive interaction

The reasoning Petri net model again takes slightly longer to load than the random Petri net model, with the difference here being more noticeable, especially with the large tree size. This difference is again due to the inclusion of models

for reasoning about the interactions, with the extra time taken here of 77 seconds outweighing the saving of 16 seconds saved by the reasoning. Comparing the total time of the reasoning Petri net model, 1137 seconds, to that of the constraint model, 1731 seconds, the Petri net is still slightly faster at performing the reasoning and also offers the better results based on plans used for these settings. However, comparing the memory requirements shown in table 6.11 again shows the Renew application simulating the Petri net model using markedly more memory than GNU Prolog for evaluating the constraint model. The reduction in the number of plans used is mirrored in the memory used for the Petri net model for the large tree size where the reduction is the greatest, however the reduction is not large enough for the smaller tree sizes to counter the additional memory required for the reasoning itself.

	Petri net model (Mb)				Constraints model (Mb)					
	Reasoning		Random		Standard		Most Constr.		Max. Regret	
	Ready	Run	Ready	Run	Ready	Run	Ready	Run	Ready	Run
Small	79.97	91.36	77.79	88.14	5.61	6.21	5.60	6.23	5.60	6.23
Med.	107.02	143.73	105.04	136.03	6.09	7.39	6.09	7.43	6.10	7.45
Large	154.77	188.04	99.43	213.51	6.86	9.22	6.86	9.26	6.87	9.30

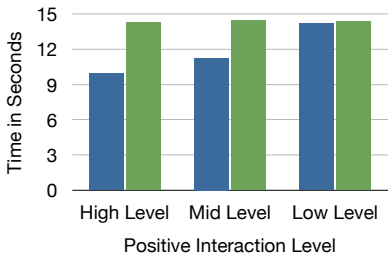
Table 6.11: Memory usage for setting: Deep tree, high level positive interaction, high goal interaction, 20 goals, varying tree size and reasoning about positive interaction

The load times and memory usage are a reflection of the file sizes for the two models, with the files for the constraint model starting at 36 Kb for the small tree size and increasing to 116 Kb for the large tree size. In comparison, the Petri net model starts at 804 Kb for the initial PNML file format containing the representation for the small goal-plan trees, which increases to 5389 Kb when imported to the Renew file format and the large tree taking 2760 Kb in the PNML file format and 18141 Kb once loaded into Renew.

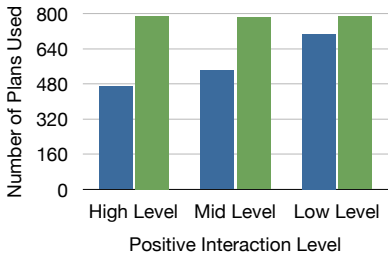
Varying Positive Interaction Level

Having looked at the effect of tree size on the positive interaction reasoning of the two models, this next set of experiments looks at the effect of varying the level at which the positive interaction occurs. This is done using the medium sized

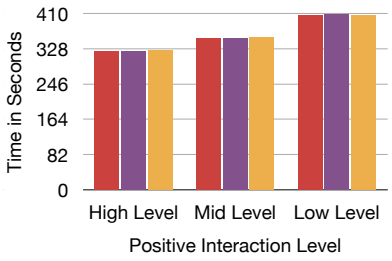
deep tree and 20 goals with high levels of goal interaction between them. This high level of interaction simply refers to the amount of interaction between the different goals, while the positive interaction level refers to the depth within the tree that the interaction actually takes place. The goals are again all designed to be achievable by both models and without reasoning, so the graphs showing goals started and achieved are omitted. The results for the Petri net and constraint model are shown in figure 6.6.



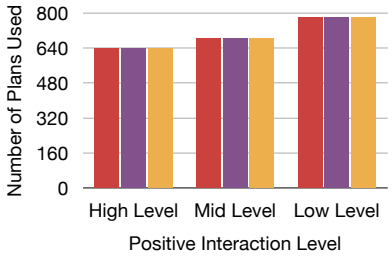
(a) Petri net Timing



(b) Petri net Plans



(c) Constraint Timing



(d) Constraint Plans

Figure 6.6: Results for setting: Medium sized deep tree, high goal interaction, 20 goals, varying positive interaction level and reasoning about positive interaction

As described above, the impact of the positive interaction is greatest when the positive interaction occurs at a high level within the goal-plan tree. When occurring at a lower level, the reduction in the number of plans and time saved is quite small, for the same reasons as those for the small tree size in the previous set of experiments. For the constraint model, the number of plans used in the experiments for interaction at a low level in the goal-plan tree was 780 compared

to the 787 used by the random Petri net, giving a saving of just 7 plans, while the reasoning Petri net was able to save an average of 79 plans with a CV of 0.6%. Clearly the saving from the constraint model does not justify the extra time taken for reasoning about the interaction at the lower level, however when the positive interaction occurs at high levels in the goal-plan tree both models, in particular the Petri net model, make considerable savings on the number of plans used.

In the experiments applying the positive interaction at a high level, the reasoning Petri net used just 471.6 plans on average, as opposed to the 789.6 plans used by the random Petri net. This leads to the reasoning Petri net using just 23.58 plans per goal, compared to the 39.48 needed by the random Petri net, an overall saving of 15.9 plans per goal, which is a significant saving. This saving reduces to 12.11 plans for the mid level, and just 3.97 plans per goal at low level using the reasoning Petri net. This saving is again mirrored in the time taken for the simulation of the Petri net model, with a reduction of 4.3 seconds for interactions at a high level in the goal-plan trees, dropping to an average saving of just 0.2 seconds for interactions at a low level, with the CV being just 5% for the timings of the experiments. The additional times for loading the models are shown in table 6.12.

	Petri net model (seconds)				Constraints model (seconds)		
	Reasoning		Random		Standard	Most	Max.
	Import	Save	Import	Save		Constr.	Regret
High	93	217	80	191	0.318	0.313	0.313
Mid	96	223	77	193	0.307	0.307	0.309
Low	111	224	81	201	0.306	0.310	0.310

Table 6.12: Load timings for setting: Medium sized deep tree, high goal interaction, 20 goals, varying positive interaction level and reasoning about positive interaction

Despite the Petri net model having a faster run time than the constraint-based model, the combined load and run times of the Petri net experiments are much slower. The slowest of these totalling to 349 seconds for the Petri net model to load and run a simulation of the interaction occurring at low levels. This is slightly slower than the total time taken for the constraint model to evaluate the interaction at high levels, 324 seconds, but faster than the evaluation of the constraints at mid and low levels in the tree, 354 and 408 seconds respectively.

	Petri net model (Mb)				Constraints model (Mb)					
	Reasoning		Random		Standard		Most Constr.		Max. Regret	
	Ready	Run	Ready	Run	Ready	Run	Ready	Run	Ready	Run
High	107.02	143.73	105.04	136.03	6.09	7.39	6.09	7.43	6.10	7.45
Mid	108.41	131.55	105.51	135.69	6.10	7.30	6.09	7.35	6.10	7.36
Low	124.00	138.58	104.99	123.64	6.09	7.15	6.10	7.19	6.10	7.22

Table 6.13: Memory usage for setting: Medium sized deep tree, high goal interaction, 20 goals, varying positive interaction level and reasoning about positive interaction

Comparing the file sizes to those produced for reasoning about resources, the positive interaction generates slightly larger files for the Petri net model, while the constraint model files are slightly smaller. This is reflected in the memory usage, shown in table 6.13, where the differences in file sizes are exaggerated between the reasoning for positive interaction in this set of experiments compared to those for the resource reasoning with 20 goals. The Petri net model for reasoning about resources required an average of 96.5 Mb of memory once loaded and waiting to run, while the average memory used for the Petri net model of positive interaction is 113.1 Mb. The run time usage is slightly closer for the two types of reasoning however this is due to the reduction in plans used in this set and the reduction in the number of goals adopted in the resource reasoning.

In the constraint-based model this difference is reversed requiring an average of 6.14 Mb of memory to load the files for resource reasoning before starting the evaluation, while only needing an average of 6.09 Mb for loading the positive reasoning constraints into memory.

6.2.3 Negative Interference

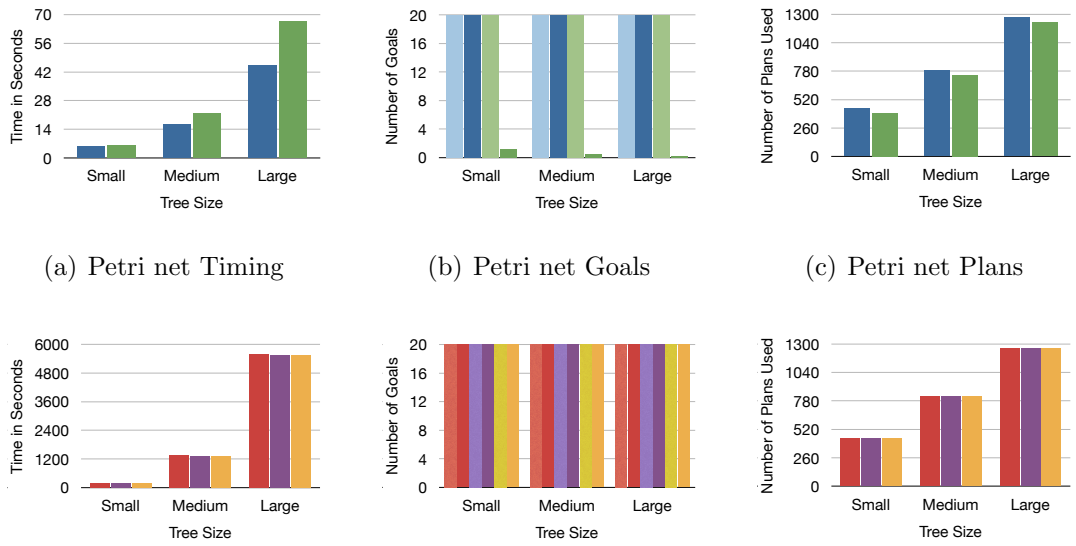
Here we now consider the results for reasoning about negative interference independently of the other two types of reasoning. The experiments for the negative reasoning were set up such that the goals were all achievable provided careful plan selection was made. These experiments illustrate the effectiveness of the reasoning under high levels of negative interference.

Varying Tree Size

As with the positive interaction, for reasoning about negative interference we start by varying the tree size using 20 top-level goals. A long duration of interference is applied, meaning that the length of time during which interference could occur is set at its maximum, as described in section 6.1.

For the deep tree in particular, there are a lot of subgoals and sub-plans between the writing and reading of the variables, and the plans for many other goals will also be attempting to write their own values to the same variables. Therefore, the reasoning must protect the variables once set until they have been read and the values can be discarded. This involves sequencing the plans such that the interference will not occur and all goals can be achieved.

The results for the Petri net and constraint-based models are shown in figure 6.7.



(a) Petri net Timing (b) Petri net Goals (c) Petri net Plans

(d) Constraint Timing (e) Constraint Goals (f) Constraint Plans

Figure 6.7: Results for setting: Deep tree, long duration negative interference, high goal interaction, 20 goals, varying tree size and reasoning about negative interference

The first thing to notice in figure 6.7(b) is the small number of goals being achieved by random Petri net. The highest success is achieved with a small tree,

where the depth is the least and therefore the duration of the interaction is also shortest. On average, the random Petri net when applied to the small tree was able to able to achieve 1.2 goals, with one repeat achieving 3 goals and two repeats achieving 2 goals out of the 20 started. As a result of this, the random Petri net also has slightly fewer plans used as it is the leaf plans where the interference is noticed so only stops a few plans from being executed. In contrast, the reasoning Petri net and constraint models consistently achieved all the goals.

The average number of plans used per goal by the reasoning Petri net were 22, 39.4 and 63.7 respectively, while the constraint model used 22, 41, 63 plans per goal respectively. The CV for the plans used in the Petri net model is 0%, 0.4% and 0.8% respectively showing that there is very little variation between the individual repeats of the experiments. The extra plans used by the constraint model for the medium sized tree is due to the slightly larger size of the first options in the branches within the tree. The number of plans are still within the range of plans required and in this set the plans themselves do not have any specific costs such as resource consumption so the extra plans do not cause any problems. In section 6.2.4, where the negative reasoning is combined with resource reasoning, it is shown that the minimum number of plans required is used in order to minimise the amount of resources consumed by individual goals.

Considering timing, the length of time taken by the random Petri net is significantly more than that required by the reasoning Petri net for each of the tree sizes, particularly for the large tree. This is due to the random nature of the Petri net unsuccessfully attempting to execute lots of different plans in the different goals, all attempting to access the variables at the same time, so constantly changing the values. This has the effect of slowing down the simulation, while the reasoning Petri net manages the order in which plans are selected, and by doing so prevents unnecessary firing of transitions in the Petri net that would otherwise slow down the simulation. The CV for the timings in both the reasoning and random Petri net is just 5% showing a very small amount of variation between the various repeats.

In the constraint model, the standard heuristic took slightly longer than the other two heuristics showing that in this reasoning there was additional backtracking caused by the constraints that could have been avoided by applying a heuristic to the model. The time difference caused in this case is only small, equating to

45 seconds between the standard and maximum regret heuristics, out of the 5598 seconds taken by the standard heuristic in total.

Comparing the timing of the Petri nets to the constraint model again reveals a large difference in the length of time taken, with the negative interference reasoning also being the slowest out of all three types of reasoning, due to the fact that all the plans in all the goals are having to be considered, whereas in the other two types either whole goals could be dropped due to insufficient resources, or sub-trees could be dropped where there is a positive interaction with another related goal. The effect of this increase in scale is most greatly felt by the constraint model, where given the large tree size with the largest number of plans and subgoals, the reasoning now takes on average 5574 seconds or 92.9 minutes to finish. The large increase in time between the medium tree size and large tree size is due to the number of additional plans between the two tree sizes. In this deep tree, moving from the small to medium tree adds 27 extra plans per goal, while moving from the medium to large tree size adds a further 46 plans per goal to reason about. As the negative interference reasoning is mainly focused on finding a safe sequence of plans to avoid interference this large jump in time taken is to be expected.

Even when including the load times shown in table 6.14, the total times for each of the tree sizes is faster for the Petri net model than for the constraint model. The rate of increase in the Petri net model is also slower than that for the constraint model so for the negative reasoning the tree sizes and number of goals involved would probably need to be quite small for the constraint model to perform better with this type of reasoning.

	Petri net model (seconds)				Constraints model (seconds)		
	Reasoning		Random		Standard	Most	Max.
	Import	Save	Import	Save		Constr.	Regret
Small	20	24	19	23	0.111	0.108	0.108
Med.	89	204	80	193	0.310	0.310	0.308
Large	349	707	338	921	0.883	0.879	0.875

Table 6.14: Load timings for setting: Deep tree, long duration negative interference, high goal interaction, 20 goals, varying tree size and reasoning about negative interference

Comparing the file sizes for the two models, the representations for the negative

interaction in both models actually produce slightly smaller files than those used by the positive interaction reasoning. As shown in table 6.15, there is a small reflection of this in the memory used once the files have been loaded and before being run, between the negative interaction here and the positive interaction when varying tree size, particularly in the reasoning Petri net model.

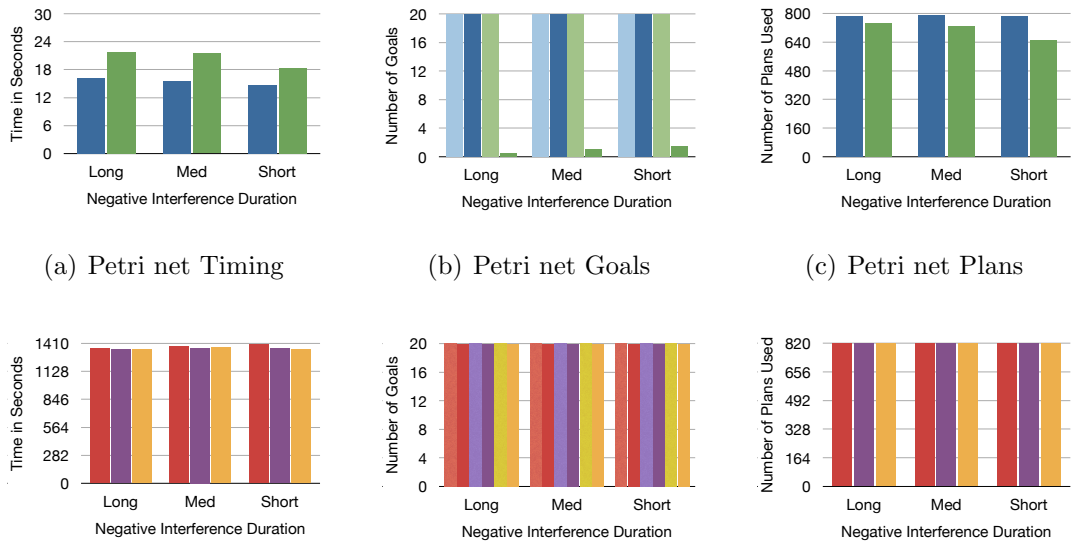
	Petri net model (Mb)				Constraints model (Mb)					
	Reasoning		Random		Standard		Most Constr.		Max. Regret	
	Ready	Run	Ready	Run	Ready	Run	Ready	Run	Ready	Run
Small	77.67	88.88	76.57	93.69	5.59	6.54	5.59	6.57	5.60	6.58
Med.	105.92	137.32	104.18	139.66	6.07	7.52	6.08	7.58	6.09	7.59
Large	152.27	213.16	153.35	212.57	6.84	8.96	6.86	9.06	6.87	9.10

Table 6.15: Memory usage for setting: Deep tree, long duration negative interference, high goal interaction, 20 goals, varying tree size and reasoning about negative interference

Varying Negative Interference Level

While in the previous set of experiments, the interference level was set to the maximum to give the longest duration of interference, in this set of experiments that level or duration is being varied by varying the height at which the interference starts within the tree, in the same way as with the positive interaction level experiments. The medium tree size is used for 20 goals, and the results are shown in figure 6.8.

Out of the 20 goals started, the random Petri net is able to attain a slight increase in the number of goals achieved as the level of negative interference decreases. However, at an average of 1.5 goals out of 20 for short periods of interference this is still a very poor result, even for the single repeat that achieved 4 goals, which shows the necessity for adding in reasoning. Interestingly, the number of plans used by the random Petri net also decreases as the level of the interaction decreases. This is a similar effect to the positive interaction, in this case the duration of the interaction is reduced by setting plans nearer to the root to read the variables, rather than moving the plans doing the writing down the tree. As a result, when the interference occurs it prevents a greater number of sub-plans from being used. This reduction in plans also leads to a reduction in the length of



(a) Petri net Timing (b) Petri net Goals (c) Petri net Plans

(d) Constraint Timing (e) Constraint Goals (f) Constraint Plans

Figure 6.8: Results for setting: Medium sized deep tree, high goal interaction, 20 goals, varying negative interference level and reasoning about negative interference

time taken for the random Petri net to complete, although still longer than that of the reasoning Petri net, which changes very little in the length of time taken to complete a simulation. This is also true for the constraint model whose timing is consistent for all three levels of negative interference as the number of plans being scheduled remains consistent.

The constraint model again uses 41 plans per goal, while the reasoning Petri net uses an average of 39.5 plans per goal across all three levels with a CV of 0.9% over the total number of plans used. While the time taken by the constraint model is again consistent between the three heuristics, the standard heuristic does show a greater time taken over the other two heuristics as was seen when varying the tree size. This variation between the three heuristics is still small compared to the total time taken for the reasoning as a whole. The loading times shown in table 6.16 again show that the total time taken for the Petri net to load and run a simulation is still significantly less than that taken by the constraint model.

As the tree sizes are all consistent the load times and in particular the memory usage, shown in table 6.17 are relatively consistent throughout and again less than

	Petri net model (seconds)				Constraints model (seconds)		
	Reasoning		Random		Standard	Most	Max.
	Import	Save	Import	Save		Constr.	Regret
Long	89	204	80	193	0.310	0.310	0.308
Med.	83	204	81	192	0.314	0.328	0.319
Short	85	201	82	199	0.306	0.305	0.309

Table 6.16: Load timings for setting: Medium sized deep tree, high goal interaction, 20 goals, varying negative interference level and reasoning about negative interference

the requirements and load times for the positive interaction models with equivalent settings.

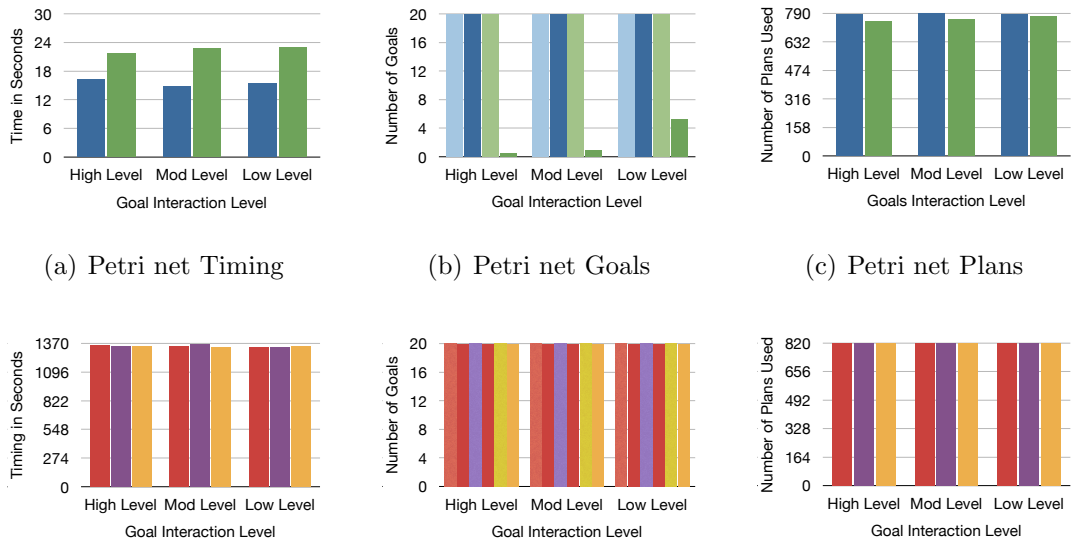
	Petri net model (Mb)				Constraints model (Mb)					
	Reasoning		Random		Standard		Most Constr.		Max. Regret	
	Ready	Run	Ready	Run	Ready	Run	Ready	Run	Ready	Run
Long	105.92	137.32	104.18	139.66	6.07	7.52	6.08	7.58	6.09	7.59
Med.	105.39	126.76	105.07	123.71	6.08	7.52	6.08	7.58	6.08	7.58
Short	105.53	126.95	104.19	125.26	6.07	7.52	6.07	7.59	6.08	7.59

Table 6.17: Memory usage for setting: Medium sized deep tree, high goal interaction, 20 goals, varying negative interference level and reasoning about negative interference

Varying Goal Interaction Level

The previous set was concerned with varying the level or duration of the negative interference. This was done at a high level of goal interaction where a lot of goals were contending for the same variables. In this set, the level at which the negative interference occurs is maintained, while the amount of interaction between goals is varied. This is accomplished by reducing the number of variables referred to by each goal so that each variable is used by a smaller number of goals, leading to a reduction in the competition between goals for a particular variable. The results for the Petri net and constraint models are shown in figure 6.9.

As with the previous set where the performance of the random Petri net increased as the duration of the interference reduced, here the performance of the random Petri net improves as the level of interaction decreases. This time however,



(a) Petri net Timing (b) Petri net Goals (c) Petri net Plans

(d) Constraint Timing (e) Constraint Goals (f) Constraint Plans

Figure 6.9: Results for setting: Medium sized deep tree, long duration negative interference, 20 goals, varying goal interaction and reasoning about negative interference

the effect is more substantial, with the random Petri net achieving an average of 5.3 goals out of 20 at low levels of goal interference. The number of goals achieved is quite consistent in the random Petri net with a range from 5-6 goals. This is still very low when compared to what is shown to be possible by using reasoning. In addition, the reasoning Petri net model, as explained previously, is in fact nearly 8 seconds faster than the random Petri net for this experimental setting, despite achieving the additional goals.

The number of plans used by the reasoning Petri net and constraint models is consistent throughout, despite the change in levels of interaction as the interaction does not affect the plans used, simply the order in which they are used. As before, the constraint model is using 41 plans per goal and the reasoning Petri net is averaging 39.5 plans per goal with a CV of 0.8%. The plans used by the random Petri net do increase slightly as the number of goals achieved increases, but still less than those required to achieve all goals.

Within the constraints model, there is very little variation in the timings again,

even between the different levels of goal interaction. As with the Petri net model, this is due to the reasoning being based on the scheduling of plans rather than on the plans used and it is this scheduling that takes the same length of time. When comparing these results to the previous set where the negative interference duration was being modified, the timings are approximately the same for this as well.

Adding in the load times for the two models, shown in table 6.18 again shows the sizes of the models and hence the load times are not changed by the variation in the level of goal interactions. The total time required for loading and simulating the reasoning Petri net is still less than the total time taken for the constraint model to find a solution.

	Petri net model (seconds)				Constraints model (seconds)		
	Reasoning		Random		Standard	Most	Max.
	Import	Save	Import	Save		Constr.	Regret
High	89	204	80	193	0.310	0.310	0.308
Med.	83	200	81	193	0.321	0.310	0.309
Low	84	195	83	195	0.306	0.313	0.305

Table 6.18: Load timings for setting: Medium sized deep tree, long duration negative interference, 20 goals, varying goal interaction and reasoning about negative interference

The memory used, (see table 6.19), shows little variation for both the standby and running memory usage when varying the amount of goal interaction. In the constraint model, there is a slight reduction in the runtime memory usage as the level of interaction reduces, indicating the reduction in the number of plans causing conflict constraints that need to be scheduled, however this does not show up in the timings for the model.

6.2.4 Combined Reasoning

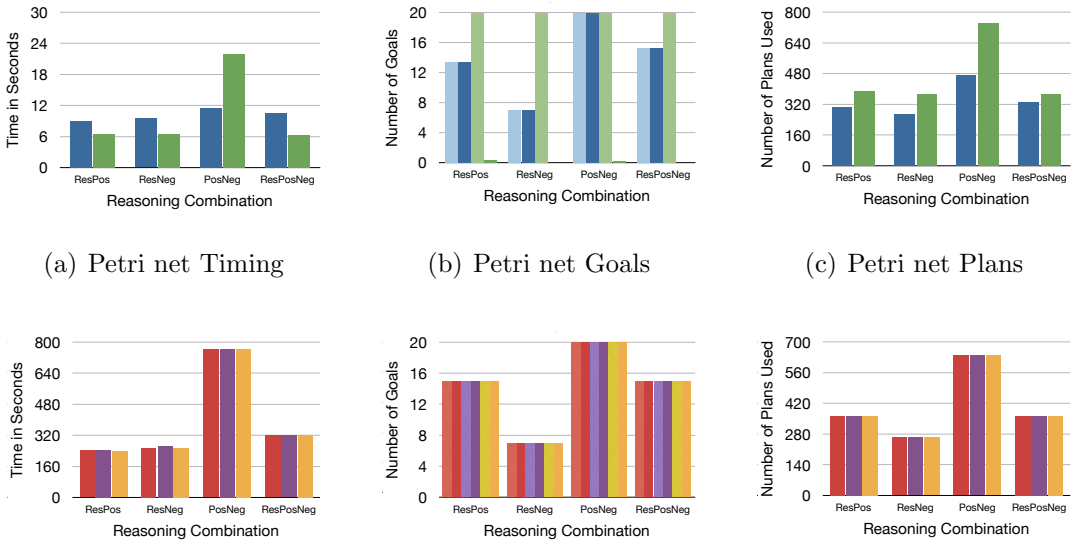
So far each of the types of reasoning have been analysed on their own to show their individual benefits and costs. By combining the reasoning, the benefits gained may be even greater than those gained using the three types of reasoning individually.

In this section we will be combining the reasoning firstly in pairs, then all three

	Petri net model (Mb)				Constraints model (Mb)					
	Reasoning		Random		Standard		Most Constr.		Max. Regret	
	Ready	Run	Ready	Run	Ready	Run	Ready	Run	Ready	Run
High	105.92	137.32	104.18	139.66	6.07	7.52	6.08	7.58	6.09	7.59
Med.	104.71	141.13	105.04	123.79	6.08	7.28	6.08	7.34	6.07	7.34
Low	106.00	135.79	104.46	123.32	6.07	7.13	6.07	7.20	6.08	7.21

Table 6.19: Memory usage for setting: Medium sized deep tree, long duration negative interference, 20 goals, varying goal interaction and reasoning about negative interference

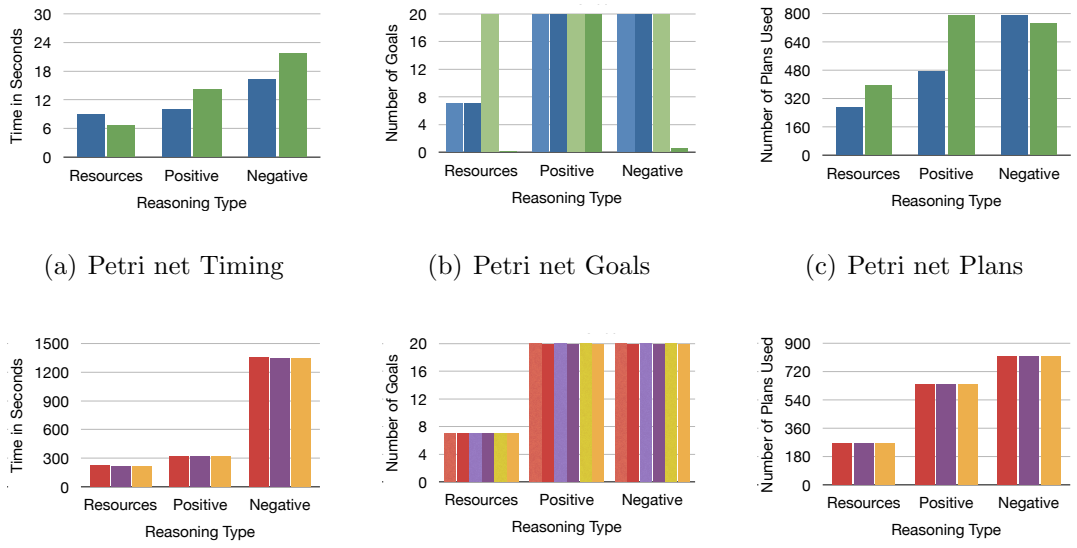
types combined together to examine the combined effects of the reasoning. A single setup has been used across all combinations, namely, 20 goals using the medium sized tree with low resource availability and high levels of positive, negative and goal interaction as described in the previous experiments. The results for the Petri net and constraint models are shown in figure 6.10, and for ease of reference the individual types of reasoning for the same settings are shown in 6.11.



(a) Petri net Timing (b) Petri net Goals (c) Petri net Plans

(d) Constraint Timing (e) Constraint Goals (f) Constraint Plans

Figure 6.10: Results for setting: Medium sized deep tree, low resource availability, high level positive interaction, long duration negative interference, high goal interaction, 20 goals, varying reasoning combinations



(a) Petri net Timing (b) Petri net Goals (c) Petri net Plans

(d) Constraint Timing (e) Constraint Goals (f) Constraint Plans

Figure 6.11: Comparison results for medium sized deep tree, individual reasoning types

The first point to note from these combined results comes from the combination of resource reasoning with negative reasoning. In terms of the reasoning Petri net this makes very little difference, however the random Petri net is unable to achieve any goals. Between running out of resources and goals interfering it is highly unlikely that a random selection will find a successful ordering of plans under these circumstances.

The most interesting combination is that of resource reasoning combined with positive interaction reasoning. As the positive reasoning reduces the number of plans needed, and some of these plans would have consumed resources this means that the overall resource requirements of a goal can drop significantly. As a result, more goals can be achieved given the same amount of resources, so instead of just achieving 7 goals, the reasoning Petri net with the combined reasoning is now able to achieve 13-15 goals using the same amount of resources. In addition to this, the random selection of goals started when all types of reasoning are included in the Petri net model allows additional goals to be started, bringing the average up to 15 goals. However, as shown by the constraint model, this is purely due to the

random order in which the goals are selected as the number of goals completed by the constraint model when resource and positive interaction reasoning are included is the same as for when all the types of reasoning are included, where the constraint model consistently achieves 15 goals.

The timing for the combination of positive and negative interaction for both the Petri nets and the constraint models are the longest as expected due to the larger number of goals and plans being used, however it is quite interesting to note that the timing for the combinations of resource & positive and resource & negative are almost identical, despite the resource & positive achieving more goals. However, when looking at the number of plans used, these are also almost identical for the Petri net as the same number of plans will consume the same amount of resources. Equally interesting is the difference between the reasoning and random Petri nets for the positive & negative combination. In this experiment, the random model is considerably slower, using substantially more plans while only achieving one goal during four repeats and an overall average of just 0.3 goals.

As with the individual types of reasoning, the Petri net model is able to make greater savings in terms of the number of plans used when reasoning about positive interaction. This does not affect the performance when combined with resource reasoning as the same number of goals are achieved even though more plans are used, as not all the plans will consume resources. In addition, the constraint model is sorting the goals into order of increasing resource requirements so more goals can be achieved. In comparison, where the Petri net is using fewer plans, the range of goals achieved for all three types of reasoning combined varies from 13 to 18, with just one repeat at each extreme. If the order in which the goals for the Petri net model were selected was sorted by resource requirements, it is possible that more goals would be consistently achieved here with the lower number of plans being used.

Comparing the timings of the various combinations, it is clear to see that the reasoning about the negative interference is the slowest of the three types of reasoning individually and slows down any combination of reasoning where it is included. While the resource & positive combination and the resource & negative combination in the constraint model seem to take roughly the same length of time, it should be noted that the first combination achieves more goals and uses slightly

more plans than the second combination.

When considering the load times for the different combinations of reasoning in the two models, as shown in table 6.20 and comparing them to those for the individual types of reasoning, as shown in table 6.21 it is clear to see that the combined reasoning produces larger files and that they take longer to load for both models. As with the individual types of reasoning in the first set of experiments, when the loading times for the resource & positive reasoning combination are included into the total simulation time for the Petri net model, then the constraint model is faster. This is repeated for the resource & negative combination; the constraint model is also slightly faster for the three types of reasoning combined together.

	Petri net model (seconds)				Constraints model (seconds)		
	Reasoning		Random		Standard	Most	Max.
	Import	Save	Import	Save		Constr.	Regret
ResPos	101	249	86	203	0.317	0.309	0.315
ResNeg	91	222	85	212	0.315	0.310	0.313
PosNeg	96	230	82	196	0.322	0.307	0.305
ResPosNeg	105	261	88	207	0.308	0.309	0.321

Table 6.20: Load timings for setting: Medium sized deep tree, low resource availability, high level positive interaction, long duration negative interference, high goal interaction, 20 goals, varying reasoning combination

	Petri net model (seconds)				Constraints model (seconds)		
	Reasoning		Random		Standard	Most	Max.
	Import	Save	Import	Save		Constr.	Regret
Res	88	212	87	207	0.307	0.306	0.310
Pos	93	217	80	191	0.318	0.313	0.313
Neg	89	204	80	193	0.310	0.310	0.308

Table 6.21: Load timings for comparison results of medium sized deep tree, individual reasoning types

Finally comparing the memory used for each combination of reasoning in the two models, shown in table 6.22, against the individual types of reasoning shown in table 6.23, the increase in the file sizes can again be seen in the increased memory requirements for the two models. When evaluating the constraint model,

the memory requirements are more pronounced as the additional constraints need to be taken into account for the two models together.

	Petri net model (Mb)				Constraints model (Mb)					
	Reasoning		Random		Standard		Most Constr.		Max. Regret	
	Ready	Run	Ready	Run	Ready	Run	Ready	Run	Ready	Run
ResPos	80.14	147.29	105.35	142.20	6.14	8.04	6.15	8.06	6.16	8.07
ResNeg	108.00	138.29	106.76	127.27	6.15	8.00	6.15	8.00	6.15	8.01
PosNeg	108.90	130.94	105.29	135.94	6.10	7.82	6.11	7.87	6.11	7.88
ResPosNeg	110.40	133.26	106.13	137.61	6.17	8.27	6.16	8.29	6.15	8.29

Table 6.22: Memory usage for setting: Medium sized deep tree, low resource availability, high level positive interaction, long duration negative interference, high goal interaction, 20 goals, varying reasoning combination

	Petri net model (Mb)				Constraints model (Mb)					
	Reasoning		Random		Standard		Most Constr.		Max. Regret	
	Ready	Run	Ready	Run	Ready	Run	Ready	Run	Ready	Run
Res	77.74	127.38	105.32	125.66	6.13	7.93	6.13	7.94	6.14	7.95
Pos	107.02	143.73	105.04	136.03	6.09	7.39	6.09	7.43	6.10	7.45
Neg	105.92	137.32	104.18	139.66	6.07	7.52	6.08	7.58	6.09	7.59

Table 6.23: Memory usage for comparison results of medium sized deep tree, individual reasoning types

6.2.5 Deep Goal-Plan Tree Conclusions

In this section we have looked at the performance of the three types of reasoning individually under different conditions to analyse and demonstrate the effectiveness of the reasoning.

While the Petri net approach can provide very fast results once loaded, they are not always optimal, and will vary slightly with each simulation. In some cases the time taken to load the Petri net model is greater than the time taken by the constraint-based model to find a solution, particularly when there is a limited amount of resources resulting in a small number of goals that can be safely adopted. The Petri net model does outperform the constraint model in terms of positive reasoning and the reduction in the number of plans used during positive

interaction in the Deep Tree, however the constraint model matches and can even outperform the number of goals achieved by the Petri net when there are limited resources available. This is most notable when resource reasoning is combined with positive interaction where the constraint model is able to sort the goals in order of resource requirement and also select the positive plan with the lowest resource requirements to keep when merging two plans.

The slowest type of reasoning for both models is that of reasoning about negative interference. This is most notable in the constraint model where the increase in time taken is considerably larger than the other two types of reasoning, especially as the number of goals and hence plans being considered increases.

When considering the memory requirements of the different types of reasoning, it can be seen that the resource reasoning requires the most memory for the constraint model, while the reasoning for the positive interaction generates the largest files in the Petri net model and hence takes the longest time to load and also requires the greatest memory

6.3 Broad Goal-Plan Trees

The broad goal-plan tree that is used for the next set of experiments (see figure 6.12) is aimed at evaluating the branch selection aspects of the two approaches, particularly in relation to the resources required and positive interaction between goals. The depth of the tree is limited to 4 levels of plans so the number of plans used when positive interaction reasoning is applied between high and low levels will not vary much. As a result, the experiments for positive interaction focus on varying the number of goals rather than the level at which the positive interaction occurred, as was done in the deep tree. As before, the experiments performed using the broad tree were selected to highlight the key features of the tree structures. In this case, the large number of branches with multiple options for selection between them.

The tree was produced such that each branch had a minimum of 2 children, with most having at least 3. The tree also had at least 3 branching plans or subgoals on each layer within the tree to form a tree that quickly expanded in width with a lot of choice within it, whilst being kept quite shallow. While in the

Size	Depth	Total Plans	Total Subgoals	Min. Plans req.	Max. Plans Req.
Small	4	24	17	13	16
Medium	4	54	39	28	34
Large	4	85	67	48	58

Table 6.24: Plan requirements for the three sizes of broad tree used

deep tree the different sizes referred to the depth of the tree, in this section the depth is kept static and the breadth of the tree is varied when varying the tree size. The large tree structure used is illustrated in figure 6.12 with markings to indicate the breadth of the medium and small sized trees. The number of plans required to achieve each individual goal in the broad tree are shown in table 6.24.

6.3.1 Consumable Resources

In this section effectiveness of reasoning about resources is considered focusing on the branching aspects of the tree when there is a limited availability of consumable resources. To evaluate this, the breadth of the tree is varied to increase the number of branches and plan options available to the reasoning models for selecting the best branches to use based on their resource requirements.

Varying the Tree Size:

This first set of experiments using the broad goal-plan tree looks at the effects of varying the tree size on the reasoning about resources. The setting for the experiments consists of 20 goals with low resource availability and high goal interaction. The results for the Petri net and constraint model are shown in figure 6.13.

The first point to note here is the number of goals achieved by the two models. While the reasoning Petri net model is averaging 4 goals from the small tree, with a range from 3 to 5, the constraint model is able to achieve 7 goals. Similarly, for the medium and large sized trees the constraint model consistently achieved 8 goals as there was insufficient additional resources to achieve any extra goals in the large sized tree. However, the Petri net model only achieved an average of 6.8 goals with a range of 6 to 7 goals over all of the repeats. The better performance

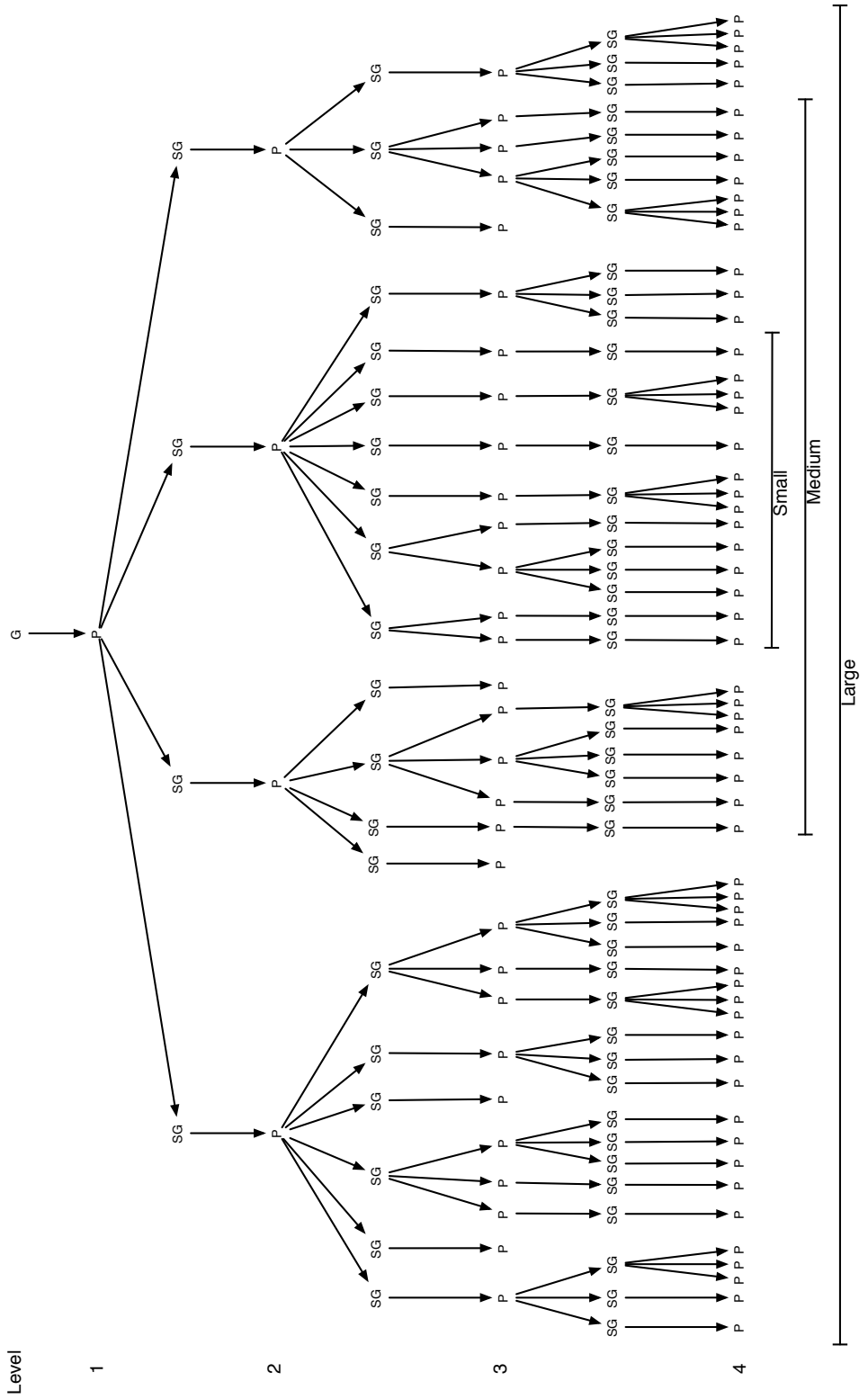
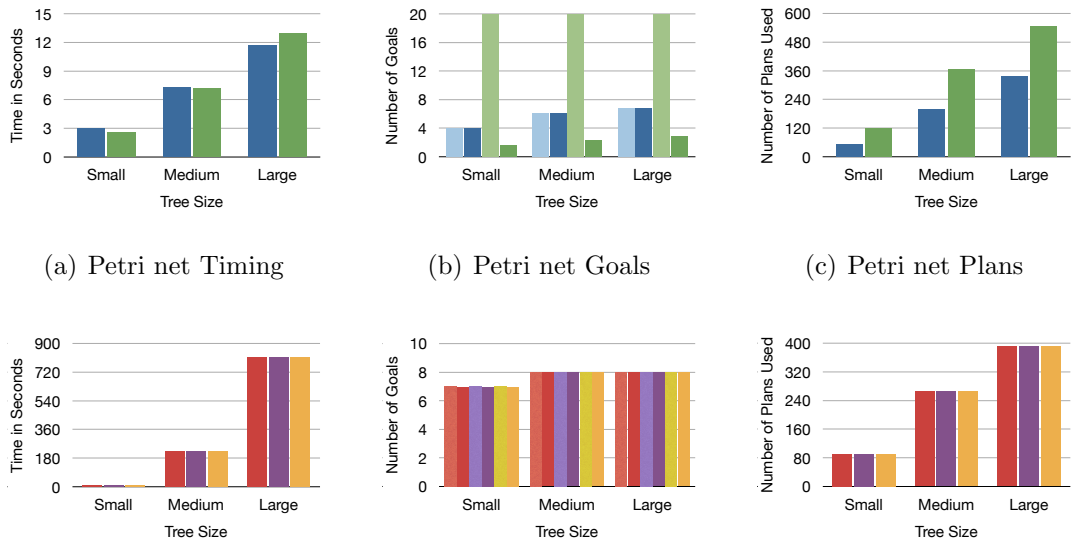


Figure 6.12: Goal-plan tree for the broad tree, showing the breadths used for small, medium and large trees



(a) Petri net Timing (b) Petri net Goals (c) Petri net Plans

(d) Constraint Timing (e) Constraint Goals (f) Constraint Plans

Figure 6.13: Results for setting: Broad tree, low resource availability, high goal interaction, 20 goals, varying tree size and reasoning about resources

here from the constraint model demonstrates the effectiveness of ordering the goals based on their resource requirements, showing the number of additional goals that can be achieved when goals with lower resource requirements are started first.

While the available resources were increased proportionally as the tree sizes were increased, the increase was not sufficient for additional goals to be achieved by the constraint model in the large size. As the tree size increases, the resource requirements for the tree increase as well. The additional resources were only able to cover the increased cost of the same number of goals with the larger tree structure, without leaving sufficient for any further goals to be started.

Comparing the goals achieved by the random Petri net for this tree structure to those achieved when applied to the deep tree structure, a greater number of goals have been successfully completed here, achieving an average of 2.4 goals with the medium sized broad tree, compared to 0.1 goals for the same settings in the deep tree. This is due to the reduced depth of the tree and increased branching resulting in random selection of branches that have lower resource requirements, which can be completed quicker than the deep plan branches in the previous tree structure.

When considering the number of plans used, the number of plans per goal used by both models was the minimum number as stated in 6.13 for the small tree size where the range of branches was the smallest. In the medium and large tree sizes, there was greater variation in the resource costs of each of the branches resulting in the cheapest branch not always containing the least plans. This is shown by the reasoning for the medium sized tree using an average of 33 plans per goal in both models, and 49 plans per goal for the large tree size in both models.

The timings for the Petri net model show that the duration taken by the reasoning Petri net to achieve its goals compared to the random Petri net, which achieved a smaller number of goals, is quite close. However, the time taken by the random Petri net increased faster than that taken by the reasoning Petri net as the tree size increased so in the large tree the Petri net model had a significant saving in the time taken compared to the random Petri net. This is highlighted in the number of plans used by each, with the random Petri net starting all the goals and executing as many plans as possible from each of them, compared to the reasoning Petri net where only goals that could be achieved were adopted, thereby limiting the selection of plans.

Comparing the time taken between the two models again shows a large difference between them which is compensated for when including the loading times, shown in table 6.25. As can be seen in both models, the time taken to load the Petri net model and to find a solution in the constraint based model increases dramatically as the tree size increases. For the Petri net model this is slightly more linear than the constraint model, increasing from a total of 53 seconds for the small tree, 511 seconds for the medium tree and 1312 seconds for the large sized tree compared to the 816 seconds required for the constraint model, up from 224 seconds for the medium sized tree and 14 seconds for the small sized tree. In each case, the constraint model is faster, however as the tree size or resource availability increases it is likely that the total duration taken by the Petri net model would be less than that taken by the constraint model.

The memory usage for the two models are shown in table 6.26 and reflect the increase in files sizes again as the tree sizes increase. The memory requirements for the constraint model increase significantly more here than for the same settings in the deep tree.

	Petri net model (seconds)				Constraints model (seconds)		
	Reasoning		Random		Standard	Most	Max.
	Import	Save	Import	Save		Constr.	Regret
Small	23	27	20	23	0.098	0.100	0.092
Med.	158	346	132	307	0.347	0.345	0.422
Large	426	875	375	783	0.697	0.701	0.698

Table 6.25: Load timings for setting: Broad tree, low resource availability, high goal interaction, 20 goals, varying tree size and reasoning about resources

	Petri net model (Mb)				Constraints model (Mb)					
	Reasoning		Random		Standard		Most Constr.		Max. Regret	
	Ready	Run	Ready	Run	Ready	Run	Ready	Run	Ready	Run
Small	78.99	96.92	77.18	88.12	5.62	6.23	5.62	6.25	5.62	6.24
Med.	121.68	146.37	112.20	140.12	6.21	7.85	6.21	7.93	6.22	7.94
Large	161.42	193.67	154.6	213.12	6.73	9.36	6.73	9.36	6.73	9.39

Table 6.26: Memory usage for setting: Broad tree, low resource availability, high goal interaction, 20 goals, varying tree size and reasoning about resources

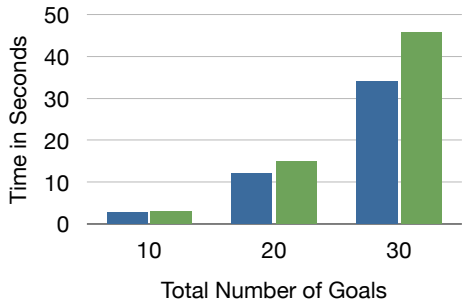
6.3.2 Positive Interaction

In the broad tree, there are only a small number of levels so varying the level at which the interaction occurs will have little affect on the number of plans used. Instead, varying the number of goals with high levels of goal interaction will provide a more suitable test to evaluate the performance of the reasoning when there is a large number of branching options to consider.

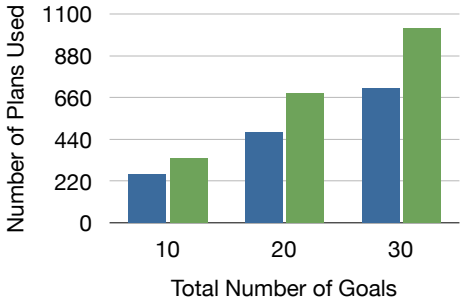
As with the deep tree, the positive interaction experiments are designed so that all goals are achievable without any reasoning. Therefore, the graphs showing the goals achieved are omitted as all the goals are always achieved by both models and the random Petri net.

Varying number of goals:

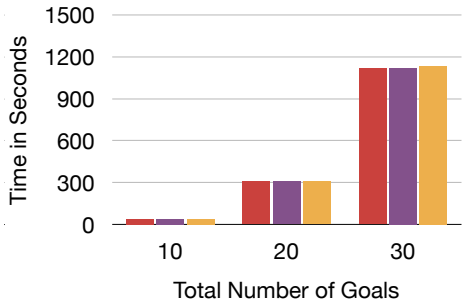
The experiments performed in this section vary the number of goals used, while using a medium sized broad tree, positive interaction at a high level within the tree and high goal interaction. The results for the Petri net and constraint models are shown in figure 6.14.



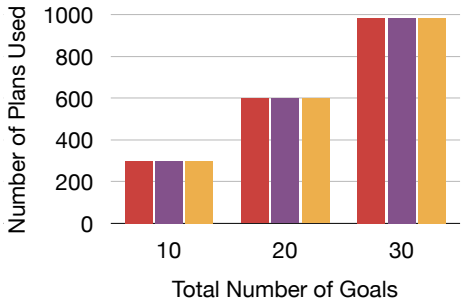
(a) Petri net Timing



(b) Petri net Plans



(c) Constraint Timing



(d) Constraint Plans

Figure 6.14: Results for setting: Medium sized broad tree, high level positive interaction, high goal interaction, varying number of goals and reasoning about positive interaction

The number of plans saved by the reasoning Petri net model is 86 plans for the small tree size (CV 2.9%), 203 plans for the medium sized tree (CV 1.6%) and 316 plans for the large tree size (CV 1.6%). Compared to this, the constraint model gives less saving, with the large tree size requiring 984 plans compared to the random Petri net requiring 1027 plans. This is just a saving of 43 plans. As with the deep tree, this is because the constraint model simply selects the first branch to keep when no other reasoning is applied. By applying a sort to interacting plan options it would be possible to select the branch with the smaller number of plans contained in it, thereby reducing the total number of plans used.

As is to be expected, the random Petri net takes longer than the reasoning

Petri net model due to the larger number of plans being used. This shows that the additional run-time cost introduced by the reasoning is negligible when compared to the savings gained from the reasoning. The major difference in the time costs comes from the loading times for the Petri net model, shown in table 6.27, showing that the reasoning Petri net does take slightly longer to import and save than the random Petri net due to the additional places and transitions included for the reasoning. This also illustrates the differences in the file sizes between the reasoning and random Petri net models. While the reasoning Petri net for the positive interaction in the deep tree was larger than that for the resource reasoning, the opposite here is true. The files for the broad tree are larger than those generated for the deep tree model, with the broad tree files being 1799 Kb while the equivalent for the deep tree model being 1534 Kb.

Comparing the combined timings for the loading and running of the two models, shown in table 6.27, shows that the total time taken for the Petri net model is again greater than that taken for the constraint model. However, the difference in timings decreases in proportion to the total time taken as the number of goals increases, starting at a difference of 42 seconds for the small tree, 191 seconds for the medium and 55 seconds for the large tree size.

	Petri net model (seconds)				Constraints model (seconds)		
	Reasoning		Random		Standard	Most	Max.
	Import	Save	Import	Save		Constr.	Regret
10	36	43	30	58	0.139	0.138	0.128
20	149	342	124	296	0.342	0.340	0.343
30	381	757	320	641	0.653	0.682	0.654

Table 6.27: Load timings for setting: Medium sized broad tree, high level positive interaction, high goal interaction, varying number of goals and reasoning about positive interaction

The memory usage of the two models is shown in table 6.28. Again the memory used for the Petri net model is greater than the memory used for the constraint model. It should also be noted that the memory required for the constraint model to reason about 30 goals is less here than that required to reason about 20 large goals with limited resource availability.

	Petri net model (Mb)				Constraints model (Mb)					
	Reasoning		Random		Standard		Most Constr.		Max. Regret	
	Ready	Run	Ready	Run	Ready	Run	Ready	Run	Ready	Run
10	86.64	99.18	85.15	103.88	5.74	6.39	5.74	6.42	5.74	6.43
20	120.22	162.07	115.00	155.25	6.16	7.32	6.15	7.35	6.17	7.37
30	155.69	188.65	144.76	196.00	6.50	8.04	6.50	8.12	6.50	8.12

Table 6.28: Memory usage for setting: Medium sized broad tree, high level positive interaction, high goal interaction, varying number of goals and reasoning about positive interaction

6.3.3 Negative Interference

As with the experiments on positive interaction reasoning for the broad tree, the focus here is on the effects of branching on the reasoning. This is evaluated by varying the number of goals.

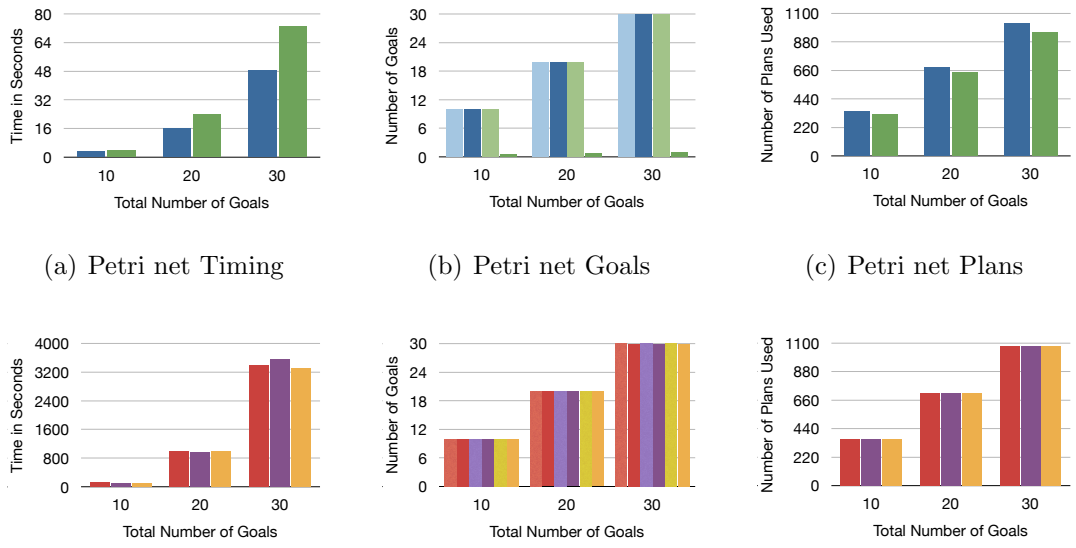
Varying number of goals:

The settings used to evaluate the negative interference reasoning on the broad tree consist of varying the number of goals with a medium tree size, long duration negative interference and high goal interaction. The results for the Petri net and constraint models are shown in figure 6.15.

As in the experiments for the deep tree, the random Petri net model is only able to achieve an average of less than 1 goal regardless of how many goals are started. This is despite the duration for which the effects need to be protected being much shorter than that required in the deep tree.

The plans used by the random Petri net model is slightly less than those used by the reasoning Petri net and the constraint models, where the Petri net model is using an average of 33.6 plans per goal, with a CV of 1% over the total number of plans used and the constraint model is using 34 plans per goal.

When considering the timings, the Most Constrained heuristic used by the constraint model is noticeably slower than the other two, adding an extra 176 seconds onto the time taken by the standard heuristic. However, with this taking over 56 minutes in total the time difference is still very small. Comparing the time taken here by the negative interference reasoning in the constraint model to that



(a) Petri net Timing (b) Petri net Goals (c) Petri net Plans

(d) Constraint Timing (e) Constraint Goals (f) Constraint Plans

Figure 6.15: Results for setting: Medium size broad tree, long duration negative interference, high goal interaction, varying the number of goals and reasoning about negative interference

in the deep tree, there is a significant reduction in the time taken by the reasoning for the broad tree, by approximately 360 seconds. This is a result of the reduced depth and so a reduction in the number of plans that need to be scheduled between the start and end points of causally linked plans.

The additional load times shown in table 6.29 show that despite the extra time taken to load the Petri net model, the total time taken by the Petri net model is still less than that taken by the constraint model to find a solution.

The file sizes produced for the Petri net model of reasoning about negative interference are smaller than those produced for the positive interaction model. As a result, the load times are slightly faster and the memory requirements for the Petri net model here are slightly lower. In the constraint model, which uses very little memory in comparison, the file sizes are the same and the reasoning requires slightly more memory whilst finding a solution.

	Petri net model (seconds)				Constraints model (seconds)		
	Reasoning		Random		Standard	Most	Max.
	Import	Save	Import	Save		Constr.	Regret
10	31	38	28	38	0.127	0.129	0.127
20	127	298	124	284	0.344	0.348	0.349
30	325	653	323	554	0.658	0.659	0.663

Table 6.29: Load timings for setting: Medium size broad tree, long duration negative interference, high goal interaction, varying the number of goals and reasoning about negative interference

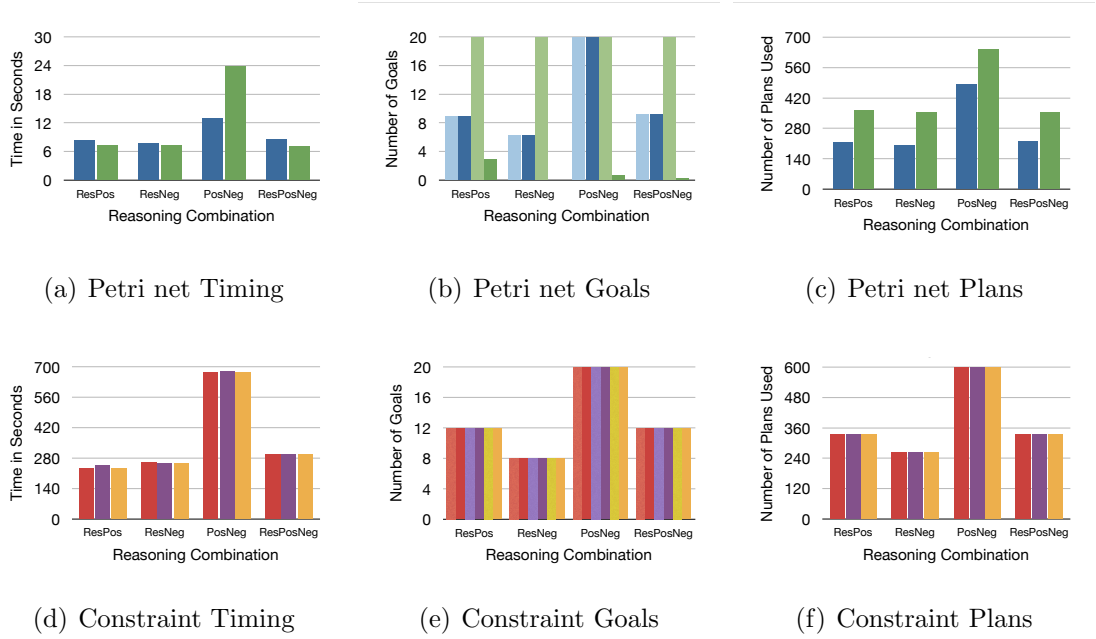
	Petri net model (Mb)				Constraints model (Mb)					
	Reasoning		Random		Standard		Most Constr.		Max. Regret	
	Ready	Run	Ready	Run	Ready	Run	Ready	Run	Ready	Run
10	85.30	97.13	84.43	96.69	5.74	6.41	5.74	6.44	5.73	6.43
20	116.23	157.60	115.08	154.50	6.16	7.57	6.16	7.61	6.15	7.61
30	149.91	183.05	155.01	182.70	6.50	8.74	6.50	8.84	6.50	8.87

Table 6.30: Memory usage for setting: Medium size broad tree, long duration negative interference, high goal interaction, varying the number of goals and reasoning about negative interference

6.3.4 Combined Reasoning

In this final section using the broad tree, the different types of reasoning that have been considered separately above are combined in each of the possible combinations to evaluate the effectiveness of the joint reasoning over the broad goal-plan tree. The settings used are again the extreme settings for each of the types of reasoning, so low resource availability, positive interaction at a high level, negative interference for a long duration, and high goal interaction. The results for the Petri net and constraint models are shown in figure 6.16. For ease of comparison, the related results for the individual types of reasoning are included in figure 6.17.

While in the deep tree the combination of resource & positive reasoning had a large impact on the number of goals that were achieved, the effect is less noticeable here especially in the Petri net model with an increase of just 2 goals out of 20 on top of the 6 goals achieved with the negative interference reasoning combined with the resource reasoning. In the constraint model, the effect of the combined reasoning is more obvious with an increase of 4 goals, over and above the higher



(a) Petri net Timing (b) Petri net Goals (c) Petri net Plans

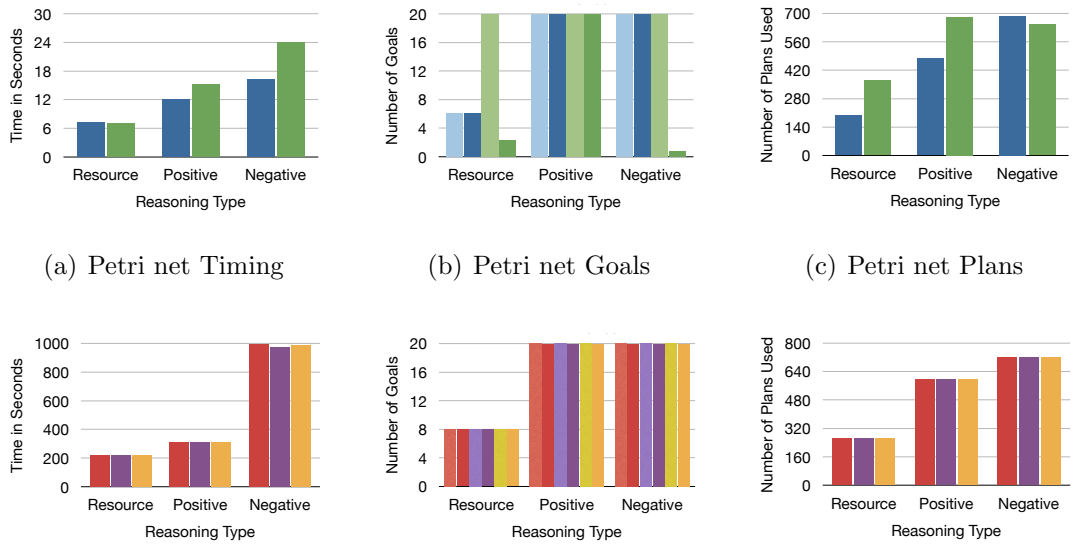
(d) Constraint Timing (e) Constraint Goals (f) Constraint Plans

Figure 6.16: Results for setting: Medium sized broad tree, low resource availability, high level positive interaction, long negative interference, high goal interaction, 20 goals, varying reasoning combination

number of goals achieved independently. Despite the additional goals achieved in both models, there is very little additional time required. In fact there is a reduction in the time taken by the constraint model despite an increase in the number of plans used when compared to the second combination comprising resource & negative interference reasoning.

The combination of positive & negative interaction reasoning is again the slowest, however it is not as slow as the negative interference reasoning on its own. This is due to the reduction in the number of plans that are considered for the final scheduling.

Adding the load times, shown in table 6.31, the first point to notice is that with the exception of the second combination of reasoning, the three combinations of reasoning for the Petri net model are relatively even in the total length of time that they add to the full reasoning time. The consequence of this is that the total length of time take to load and simulate the Petri net model is slower than that of the constraint model in all the combinations except the positive & negative



(a) Petri net Timing (b) Petri net Goals (c) Petri net Plans

(d) Constraint Timing (e) Constraint Goals (f) Constraint Plans

Figure 6.17: Comparison results for medium sized broad tree, individual reasoning types

combination. The Petri net model does however have a greater reduction in the number of plans required when positive reasoning is incorporated.

	Petri net model (seconds)				Constraints model (seconds)		
	Reasoning		Random		Standard	Most	Max.
	Import	Save	Import	Save		Constr.	Regret
ResPos	190	414	135	305	0.340	0.349	0.343
ResNeg	162	370	132	303	0.349	0.345	0.344
PosNeg	151	440	135	349	0.353	0.350	0.349
ResPosNeg	188	420	132	312	0.347	0.348	0.356

Table 6.31: Load timings for setting: Medium sized broad tree, low resource availability, high level positive interaction, long negative interference, high goal interaction, 20 goals, varying reasoning combination

Comparing the load times to the those for the individual types of reasoning, shown in table 6.32, it is clear in the Petri net model that the size of the files and hence the load times have increased with the additional reasoning being incorporated into them.

	Petri net model (seconds)				Constraints model (seconds)		
	Reasoning		Random		Standard	Most	Max.
	Import	Save	Import	Save		Constr.	Regret
Res	158	346	132	307	0.347	0.345	0.422
Pos	149	342	124	296	0.342	0.340	0.343
Neg	127	298	124	284	0.344	0.348	0.349

Table 6.32: Load timings for comparison results of medium sized broad tree, individual reasoning types

Finally, the memory usage of the different combinations, shown in table 6.33, reflects the file sizes in the loading times, with the resource & positive reasoning pair having the largest file size out of all the pairs for the Petri net model. This also follows through to the memory used when running the simulations and evaluating the constraints with the first combination having the highest memory usage of the pairs of combinations. When they are all combined together, the memory used by the Petri net model drops slightly while the constraint model uses a little more than previously.

	Petri net model (Mb)				Constraints model (Mb)					
	Reasoning		Random		Standard		Most Constr.		Max. Regret	
	Ready	Run	Ready	Run	Ready	Run	Ready	Run	Ready	Run
ResPos	126.86	172.14	119.55	141.66	6.23	8.01	6.23	8.03	6.23	8.04
ResNeg	122.32	164.67	116.69	140.64	6.23	8.00	6.22	8.01	6.22	8.02
PosNeg	123.27	164.05	115.33	138.79	6.18	7.75	6.18	7.80	6.18	7.80
ResPosNeg	127.32	152.77	116.53	140.27	6.23	8.20	6.25	8.23	6.24	8.23

Table 6.33: Memory usage for setting: Medium sized broad tree, low resource availability, high level positive interaction, long negative interference, high goal interaction, 20 goals, varying reasoning combination

A comparison of the memory requirements for the individual types of reasoning, shown in table 6.34, shows the increased memory cost when the types of reasoning are combined together.

6.3.5 Broad Goal-Plan Tree Conclusions

In this section the effectiveness and performance of the different types of reasoning have been evaluated using a broad goal-plan tree structure. The experiments

	Petri net model (Mb)				Constraints model (Mb)					
	Reasoning		Random		Standard		Most Constr.		Max. Regret	
	Ready	Run	Ready	Run	Ready	Run	Ready	Run	Ready	Run
Res	121.68	146.37	112.2	140.12	6.21	7.85	6.21	7.93	6.22	7.94
Pos	120.22	162.07	115.00	155.25	6.16	7.32	6.15	7.35	6.17	7.37
Neg	116.23	157.60	115.08	154.50	6.16	7.57	6.16	7.61	6.15	7.61

Table 6.34: Memory usage for comparison results of medium sized broad tree, individual reasoning types

performed here have shown that the constraint-based approach has achieved better results when reasoning about resources than the Petri net model, and solutions have been found in a shorter length of time.

In comparison to this, the Petri net model has achieved the better results when reasoning about positive interaction, finding greater reductions in the number of plans needed. Despite this, the overall time taken from loading to completing a simulation in the Petri net model is still slightly slower than that of the constraint model. However, the changes in time taken as the size of the problem increases suggests that the Petri net model may overtake the constraint model. In the negative interference reasoning experiments, the Petri net model is the faster of the two models, with both models able to schedule plans such that all goals can be achieved.

Finally, when all three types of reasoning are combined, the reasoning about resources and positive interactions provides a slight increase in the number of goals achieved, especially in the constraint model. The constraint model is also able to find the solutions faster than the Petri net model, when the loading times for this model are taken into consideration.

6.4 General Goal-Plan Tree

This tree was developed for the purpose of testing the performance of the two reasoning approaches with a tree that was somewhere between the deep and broad trees in formation. The tree had more branching than the deep tree, whilst having more depth than the broad tree. The main purpose of this being to test the two approaches with all the reasoning types together using a large tree size and large

Size	Depth	Total Plans	Total Subgoals	Min. Plans req.	Max. Plans Req.
Large	5	93	47	19	45

Table 6.35: Plan requirements for the general tree, only large size is used

numbers of goals to see how well they scaled up.

The tree structure used is shown in figure 6.18. Only the full tree size is used in the experiments in this section. The plan requirements for the tree are shown in table 6.35.

6.4.1 Varying the Combined Reasoning Types

As the main purpose for using this tree structure is to see how the three types of reasoning combined together scale when applied to an increasing number of large goals, this set of results starts by comparing the individual types of reasoning before they are combined together so as to provide a baseline for the experiments combining them. The settings used here are the extreme settings for each of the types of reasoning. That is, low resource availability, positive interaction at high levels in the goal-plan tree, negative interference over a long duration, high goal interaction and 20 goals. The combinations of reasoning are varied starting with pairs of reasoning types then combining all three types together. The results for the individual types of reasoning are shown in figure 6.19, with the combined reasoning shown in figure 6.20.

Firstly considering the individual types of reasoning, the number of goals achieved by the constraint model when reasoning about resources is again greater than that achieved by the Petri net model, as was the case for the broad tree. In this case, the number of goals achieved by the reasoning Petri net model is less than half that achieved in the constraint model and only slightly greater than the number achieved by the random model. The reasoning Petri net model achieved an average of 3.2 goals with a range from 3 to 4, while the random Petri net achieved an average of 1.8 goals with a range from 0 to 3. This is again related to the random order in which the goals are started in the Petri net model, whereas

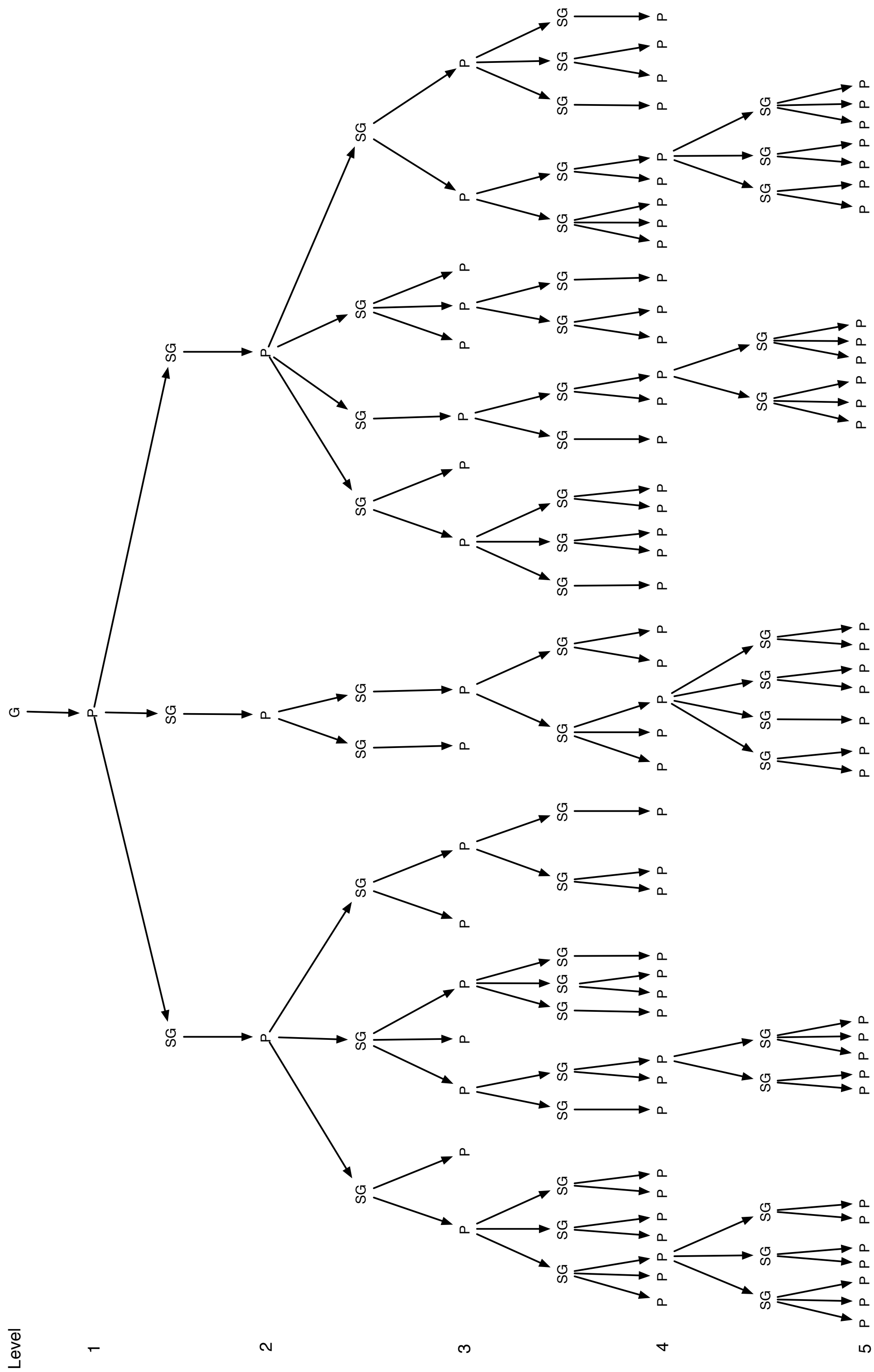
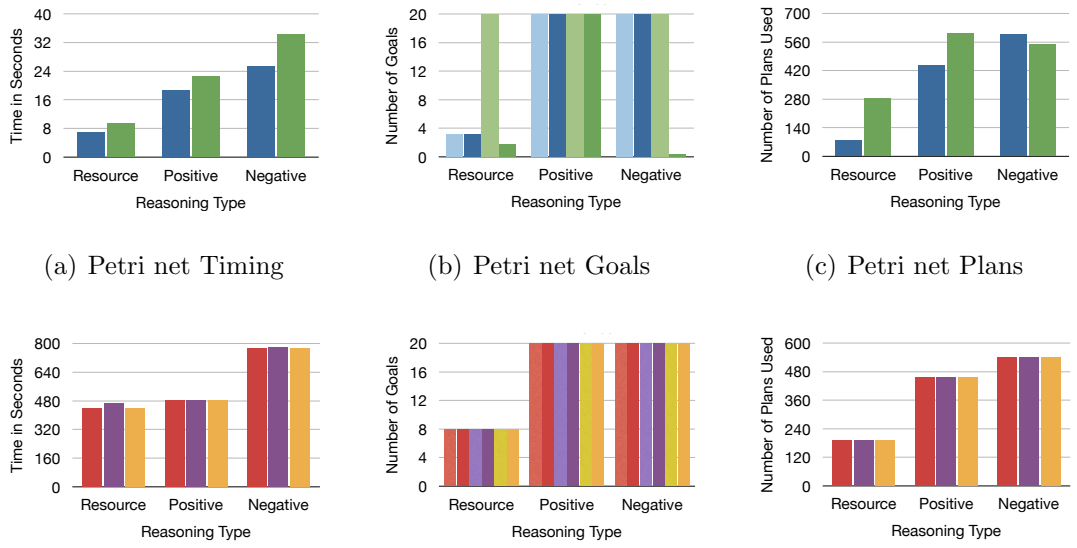


Figure 6.18: Goal-plan tree for the general tree used, showing the large tree structure



(a) Petri net Timing (b) Petri net Goals (c) Petri net Plans

(d) Constraint Timing (e) Constraint Goals (f) Constraint Plans

Figure 6.19: Comparison results for large sized general tree, individual reasoning types

the constraint model is able to sort the goals into an order based on the resource requirements of each goal.

In comparison, the Petri net model finds greater savings over the number of plans needed when reasoning about positive interaction than the constraint model. The random Petri net model used an average of 602 plans (CV 2.3%, 30 plans per goal), while the reasoning Petri net was able to bring this down to an average of 447.5 plans (CV 3.8%, 22.4 plans per goal) giving an average saving of 154.4 plans. Interestingly, this is still greater than the minimum number of plans required to achieve each of the goals, showing that the random selection of plans is inefficient. In the constraint model, this is even more noticeable, with the reduction in plans required just being 142 plans, giving an average of 23 plans per goal. This is again due to the positive interaction reasoning in the constraint model selecting the first available branch to keep when selecting branch options or merging plans, rather than considering the number of plans in each branch.

The negative interference reasoning is again the slowest of the three types of reasoning in both of the models, however without the reasoning there, the random

Petri net is only able to achieve an average of 0.3 goals with a range from 0 to 1 goals across all of the repeats. Comparing the time taken by the negative interference reasoning in the constraint model to the times taken in the deep and broad trees, there is a significant reduction, especially as the tree size being considered here is much larger. This is due to the amount of branching and sub-tree size in this tree structure reducing the total number of plans required to achieve each goal to the point where the number of plans used in total here is slightly less than that required in the previous two trees.

	Petri net model (seconds)				Constraints model (seconds)		
	Reasoning		Random		Standard	Most	Max.
	Import	Save	Import	Save		Constr.	Regret
Res	546	1089	373	785	0.590	0.601	0.638
Pos	408	837	376	883	0.824	0.658	0.650
Neg	373	776	378	768	0.596	0.595	0.656

Table 6.36: Load timings for comparison results of large sized general tree, individual reasoning types

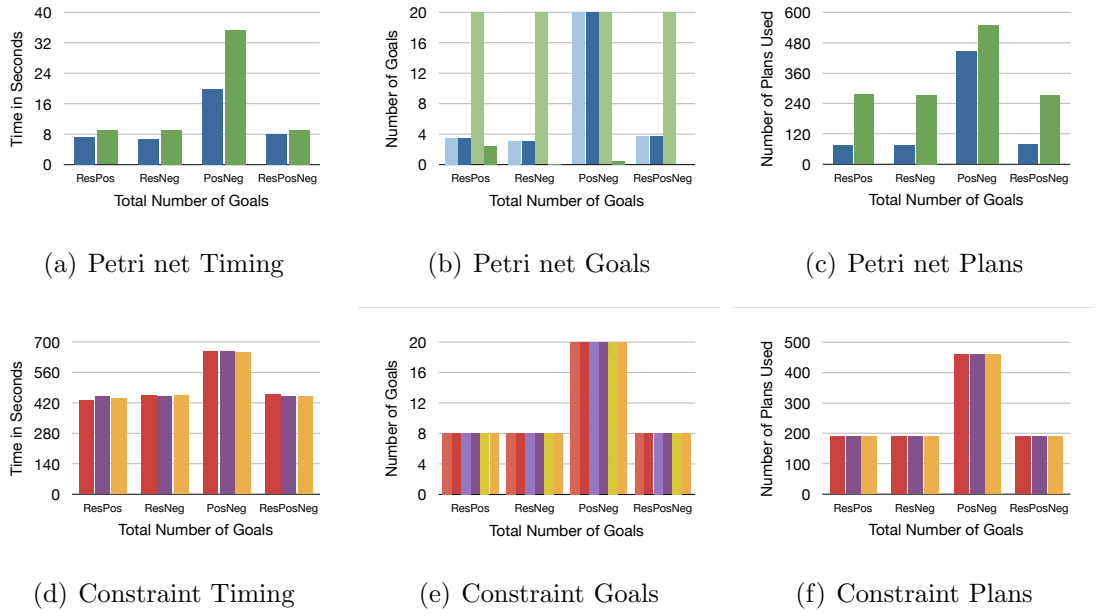
Including the load times, shown in table 6.36 into the total time taken by the two models shows that the load time for the Petri net model in particular is more substantial with this large sized tree, especially for the resource reasoning where the largest file sizes are produced. Despite the files for resource reasoning in the constraint model being slightly larger, the positive reasoning model takes a little longer to load. In total, the time taken by the Petri net model is significantly greater than that taken by the constraint model, even when reasoning about negative interference within this tree structure.

	Petri net model (Mb)				Constraints model (Mb)					
	Reasoning		Random		Standard		Most Constr.		Max. Regret	
	Ready	Run	Ready	Run	Ready	Run	Ready	Run	Ready	Run
Res	172.97	238.78	157.88	188.18	6.59	8.48	6.58	8.47	6.59	8.49
Pos	161.57	224.68	159.18	213.17	6.54	7.74	6.54	7.78	6.54	7.79
Neg	156.96	216.43	155.36	186.22	6.52	8.04	6.54	8.09	6.53	8.09

Table 6.37: Memory usage for comparison results of large sized general tree, individual reasoning types

Comparing the memory usage for the two models, it is again the Petri net

simulations that require the greatest memory, and the larger file size of the resource reasoning is reflected in the memory required to run the simulations.



(a) Petri net Timing (b) Petri net Goals (c) Petri net Plans

(d) Constraint Timing (e) Constraint Goals (f) Constraint Plans

Figure 6.20: Results for setting: Large sized general tree, low resource availability, high level positive interaction, long duration negative interference, high goal interaction, 20 goals, varying reasoning combinations

Comparing the individual types of reasoning to the combined reasoning shown in figure 6.20, the first point to notice is the combinations of resource and positive reasoning. In both models, the number of goals is not increased unlike the combined effects when applied to the deep tree. In the Petri net model the average number of goals achieved increases slightly to 3.5, however the savings from the plans not used due to positive interaction are not sufficient for either of the models to achieve additional goals for this large tree size.

As expected, the combination of positive & negative interaction reasoning is the slowest, however this is partly due to all the goals being achieved compared to just a small number, and in the constraint model this difference is proportionally smaller to that in the Petri net model. When the negative interference reasoning is incorporated with the resource & positive reasoning, the increase in cost for both models is very small.

	Petri net model (seconds)				Constraints model (seconds)		
	Reasoning		Random		Standard	Most	Max.
	Import	Save	Import	Save		Constr.	Regret
ResPos	584	1561	403	796	0.591	0.604	0.647
ResNeg	530	1634	376	1007	0.659	0.601	0.593
PosNeg	425	1003	366	875	0.591	0.598	0.605
ResPosNeg	614	1231	387	803	0.595	0.650	0.602

Table 6.38: Load timings for setting: Large sized general tree, low resource availability, high level positive interaction, long duration negative interference, high goal interaction, 20 goals, varying reasoning combinations

As shown in table 6.38, when comparing the load times to those for the individual types of reasoning, the time taken has increased considerably for each of the combinations, especially when all three types of reasoning are combined together. Interestingly, the combination of all three types of reasoning takes less time to save than the two pairings containing resource reasoning.

	Petri net model (Mb)				Constraints model (Mb)					
	Reasoning		Random		Standard		Most Constr.		Max. Regret	
	Ready	Run	Ready	Run	Ready	Run	Ready	Run	Ready	Run
ResPos	181.90	250.53	158.17	189.57	6.61	8.52	6.61	8.52	6.60	8.51
ResNeg	176.62	204.66	160.46	217.02	6.59	8.54	6.60	8.55	6.60	8.55
PosNeg	163.98	226.90	160.78	216.14	6.56	8.19	6.54	8.21	6.55	8.22
ResPosNeg	183.01	211.62	158.43	215.97	6.61	8.57	6.61	8.57	6.61	8.58

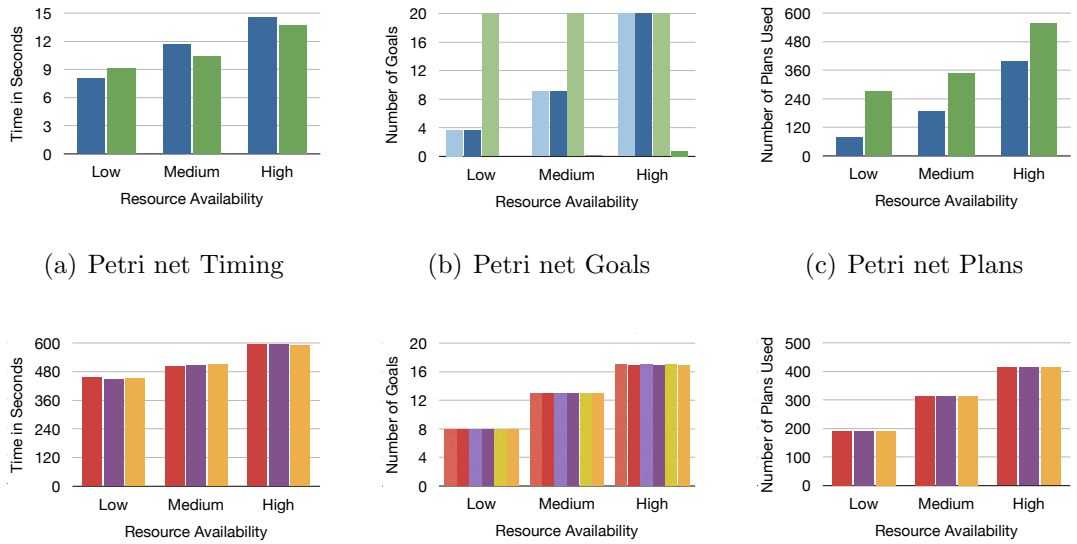
Table 6.39: Memory usage for setting: Large sized general tree, low resource availability, high level positive interaction, long duration negative interference, high goal interaction, 20 goals, varying reasoning combinations

Table 6.39 shows the memory requirements for the different combinations of reasoning. In the Petri net model, the resource & positive interaction reasoning types use the most memory, while in the constraint model the complete combination uses slightly more than the other combinations to find a solution.

Varying Resource Availability:

In the previous set of experiments a low level of resource availability was maintained for all the goals. However, when considering scaling, it is preferable to have

as many goals as possible being adopted to evaluate the scaling ability of the two models. To this end, this set of experiments varies the resource availability up to a high level of availability so that in the final set the high resource availability can be applied as the number of goals increases to stress test the two models. The results for the Petri net and constraint models are shown in figure 6.21.



(a) Petri net Timing (b) Petri net Goals (c) Petri net Plans

(d) Constraint Timing (e) Constraint Goals (f) Constraint Plans

Figure 6.21: Results for setting: Large sized general tree, high level positive interaction, long duration negative interference, high goal interaction, 20 goals, varying resource availability and reasoning about all types

While at low levels of resource availability the Petri net model fails to achieve as many goals as the constraint model, this is reversed when the resource availability is high and positive interaction reasoning is still included. This is because the total number of plans saved increases as the availability of resources rises and more goals are being safely adopted. At the high level of availability, sufficient plans are saved to provide the necessary resources to complete the extra goals resulting in all goals being achieved by the Petri net model. In comparison, the constraint model is still unable to save sufficient resources so only achieves 17 out of the 20 goals. Despite the increased number of goals and plans being used between the two models, the time taken does not increase that much.

Looking at the additional times for loading the two models, shown in table 6.40, the changes to the resource availability make very little difference to the total times taken. For the Petri net model, the total time taken is still greater than that required by the constraint model to find a solution at high levels of availability, however the Petri net model does achieve additional goals using slightly less plans at the high level of resources availability.

	Petri net model (seconds)				Constraints model (seconds)		
	Reasoning		Random		Standard	Most	Max.
	Import	Save	Import	Save		Constr.	Regret
Low	614	1231	387	803	0.595	0.650	0.602
Med.	594	1252	380	799	0.594	0.599	0.601
High	639	1246	391	802	0.588	0.593	0.593

Table 6.40: Load timings for setting: Large sized general tree, high level positive interaction, long duration negative interference, high goal interaction, 20 goals, varying resource availability and reasoning about all types

The memory usage, shown in table 6.41, shows the increase in memory required by the constraint model as the number of goals being adopted increases due to the increased availability of resources. There is also an increase in the runtime memory required by the Petri net model taking into consideration all the additional goals that are being executed.

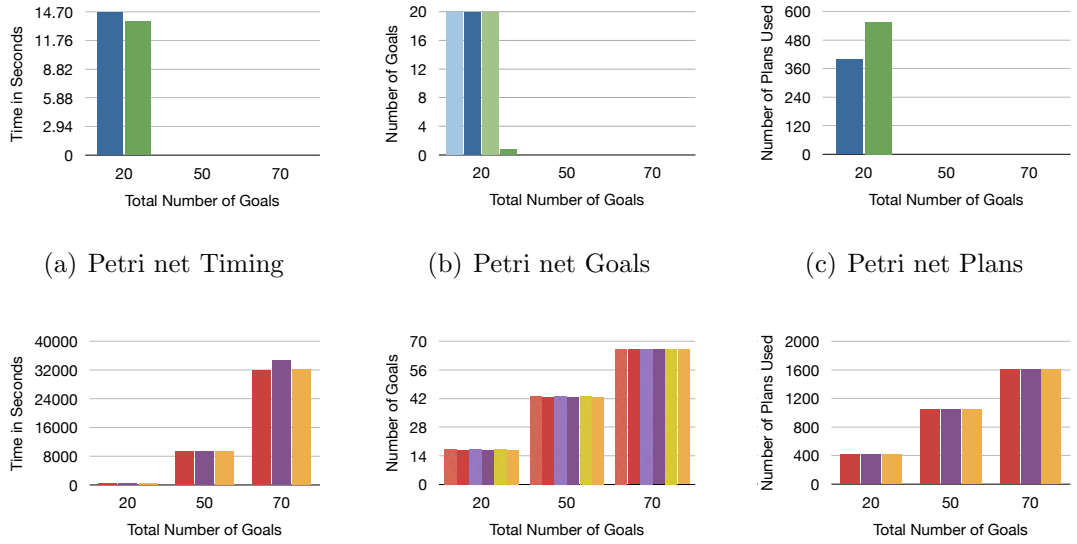
	Petri net model (Mb)				Constraints model (Mb)					
	Reasoning		Random		Standard		Most Constr.		Max. Regret	
	Ready	Run	Ready	Run	Ready	Run	Ready	Run	Ready	Run
Low	183.01	211.62	158.43	215.97	6.61	8.57	6.61	8.57	6.61	8.58
Med.	181.24	220.83	159.82	208.29	6.61	8.76	6.60	8.78	6.60	8.78
High	183.67	245.52	157.94	216.99	6.61	9.11	6.61	9.14	6.61	9.15

Table 6.41: Memory usage for setting: Large sized general tree, high level positive interaction, long duration negative interference, high goal interaction, 20 goals, varying resource availability and reasoning about all types

Varying number of goals:

In this final set of experiments we stretch the two models to see how far they can go. This is done using the large tree size with an increasingly large number of

goals at high resource availability, positive interaction at a high level in the goal-plan tree, negative interference for a long duration and high goal interaction. The results for the Petri net and constraint model are shown in figure 6.22.



(a) Petri net Timing (b) Petri net Goals (c) Petri net Plans

(d) Constraint Timing (e) Constraint Goals (f) Constraint Plans

Figure 6.22: Results for setting: Large sized general tree, high resource availability, high level positive interaction, long duration negative interference, high goal interaction, varying number of goals and reasoning about all types

The most obvious point to first notice here is the limitations of the Petri net model. In this set of experiments the large tree size was used to see how far the models could be expanded, and the Petri net model reached its limit. The Java based simulator, Renew, used to develop and run the Petri nets was unable to load the Petri nets used to represent the 50 goals being evaluated here. It is quite likely that in the future, updates to this editor or new editors will increase this limit allowing more goals using this model to be represented. It is also possible that the way goals are represented could be optimised or modified allowing current editors to handle a greater number of goals. These are all aspects that will be considered in the future work developing this model.

In chapter 4, section 4.6, different approaches to representing the Petri nets were discussed. The approach that was finally chosen was to store all the goals

individually within a single file. With the large number of large sized goals being evaluated here the editor simply was not able to handle the size of the net so failed to even import the goals definition. It is possible that if the goals had each been stored in separate files, or if one file had been instantiated 75 times, that this problem may have been avoided, however attempting to load 75 separate goal files could well have caused a different set of problems.

In comparison to this, the performance of the CSP was impressive with its ability to reason about 75 goals. The downside is that this took over 9 hours to complete, and it is predicted that 100 goals would take approximately 24 hours. Given a scenario where time was not an important factor, but being able to reason about large numbers of goals with consistent results was very important, then the CSP model would provide a viable option for performing that reasoning.

The load times shown in table 6.42, present the increased time taken for the constraint model to load as the number of goals increases. This is still very small compared to the length of time required to load the Petri net model. It is predicted that if the Petri net model had been able to load the goals, then the total time to load and run a simulation would have been less than the total time required by the constraint model as the increase in time for loading the Petri net model would be less than that for evaluating the constraint model.

	Petri net model (seconds)				Constraints model (seconds)		
	Reasoning		Random		Standard	Most	Max.
	Import	Save	Import	Save		Constr.	Regret
20	639	1246	391	802	0.588	0.593	0.593
50	-	-	-	-	1.282	1.245	1.269
75	-	-	-	-	3.674	3.668	3.681

Table 6.42: Load timings for setting: Large sized general tree, high resource availability, high level positive interaction, long duration negative interference, high goal interaction, varying number of goals and reasoning about all types

Finally, the memory requirements for the constraint model, shown in table 6.43, illustrate the increase in the memory required for the reasoning as the number of goals increases.

	Petri net model (Mb)				Constraints model (Mb)					
	Reasoning		Random		Standard		Most Constr.		Max. Regret	
	Ready	Run	Ready	Run	Ready	Run	Ready	Run	Ready	Run
20	183.67	245.52	157.94	216.99	6.61	9.11	6.61	9.14	6.61	9.15
50	-	-	-	-	7.62	12.65	7.62	12.73	7.62	12.79
75	-	-	-	-	8.49	15.66	8.50	15.78	8.50	15.81

Table 6.43: Memory usage for setting: Large sized general tree, high resource availability, high level positive interaction, long duration negative interference, high goal interaction, varying number of goals and reasoning about all types

6.4.2 General Goal-Plan Tree Conclusions

In this section, the two models have been evaluated to consider how they perform with a large tree size and how well they scale to handle an increasing number of goals when all three types of reasoning are incorporated. Within this tree structure, the constraint model has performed better overall in terms of scaling and goals achieved when there are low levels of resource availability. However, the Petri net model has still performed better when resource availability has been high and when reasoning about positive interaction. The longer loading times for the Petri net model to import and save the large tree sizes have meant that despite any savings on running time from reduced plans being used, the total time taken by the Petri net model has been greater than that required by the constraint model to find a solution. However, if the execution time for the plans is included, the load time may be negligible. This is most noticeable when reasoning about resources where the files produced for the Petri net model have been the largest leading to increased load times and memory usage.

6.5 Summary of Comparison of Tree Structures and Reasoning Models

In this chapter, the results from a wide set of experiments evaluating the performance of the three types of reasoning in the two models have been presented. The situations and conditions under which each model has been tested has highlighted areas where one model or the other performs better.

To compare the performance of the three types of reasoning on the different tree structures, a series of graphs, combining the results for common settings in each of the tree structures discussed above for each type of reasoning, are presented here for ease of reference. These show the results for experiments using medium sized deep and broad tree or large tree from the general tree structure, 20 goals, low level resource availability, positive interaction at a high level, negative interference over a long duration and high goal interaction. As the three heuristics for the constraint-based model have all given very similar timings, only the standard heuristic is presented here. In addition, when showing the timings, the load timings for both models are included in the graphs rather than separately in a table. The legend for the graphs below is shown in figure 6.23.

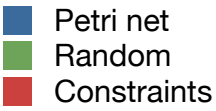
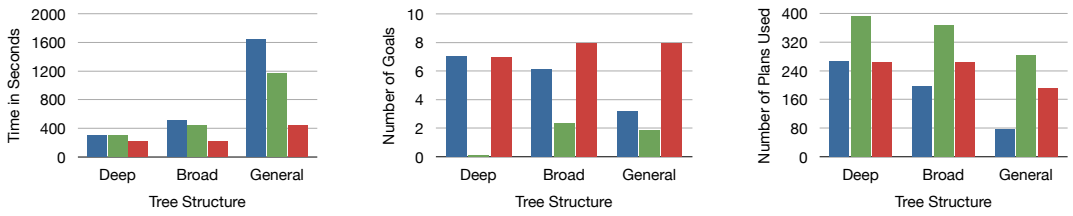


Figure 6.23: Legend for graphs comparing performance over the three different tree structures

6.5.1 Reasoning about Consumable Resources

While the Petri net model was able to match the number of goals achieved by the constraint model in the deep tree, the performance in the broad and general trees was much worse, see figure 6.24. In comparison, the random Petri net model was able to achieve more goals in the broad and general trees than in the deep tree. The timings for the Petri net model were greater than those for the constraint model when including loading times, especially in the large sized general tree structure experiments. Overall, the constraint model gave the better results both in terms of time and number of goals achieved when there is a limited availability of consumable resources, especially in trees where there is a large amount of branching.



(a) Resource Timing (b) Resource Goals (c) Resource Plans

Figure 6.24: Comparison results for reasoning about resources across the three tree structures

6.5.2 Reasoning about Positive Interaction

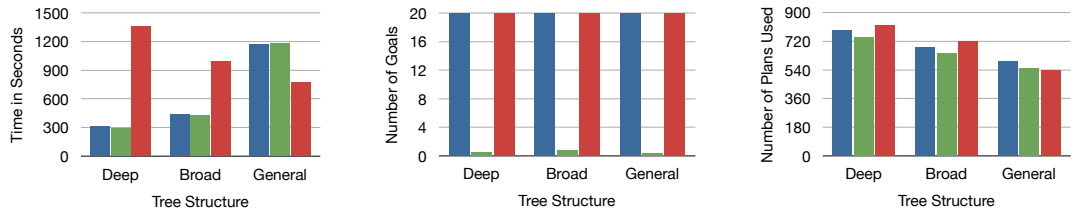
When reasoning about positive interaction, the Petri net was able to generate better results based on the reduction in the number of plans used in each of the tree structures, see figure 6.25. Comparing the timings here shows that while the time taken between the Petri net and the constraint models was the same for the deep tree, the Petri net model did take longer to load in the experiments for the other two tree structures, especially the large tree size of the general tree structure. Where the number of plans used is the key criteria then the Petri net model performs better, however if time is critical then the constraint model can produce results slightly faster when reasoning about positive interactions is desired.

(a) Positive Timing (b) Positive Plans

Figure 6.25: Comparison results for reasoning about positive interaction across the three tree structures

6.5.3 Reasoning about Negative Interference

While the reasoning about negative interference was the most time consuming of all the three types of reasoning, it is perhaps the most critical when comparing the results achieved to those produced when no reasoning is included, as illustrated in figure 6.26. In this case, the time taken by the Petri net even when the load times are included is much shorter for the experiments on the deep and broad tree structures. However, the loading time on the large sized tree for the general tree structure does take longer than the constraint model in this setting. Overall the Petri net model offers the better results here, especially with the small and medium tree structures.



(a) Negative Timing (b) Negative Goals (c) Negative Plans

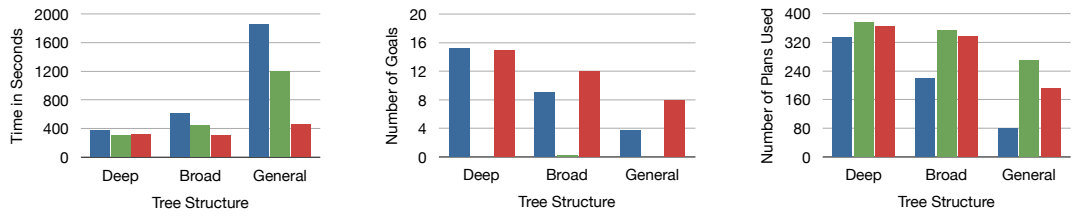
Figure 6.26: Comparison results for reasoning about negative interference across the three tree structures

6.5.4 Combined Reasoning

When combining the three types of reasoning together, the number of goals achieved increased, especially in the deep tree where a large number of plans were saved by the positive interaction reasoning, as shown in figure 6.27. The resources that would have been consumed by these plans were then available for use in achieving other goals. This combined effect is less noticeable in the broad and general trees. However, the constraint model was generally able to make the most optimisations here. The exception to this is as the availability of the resources was increased in the general tree structure, the number of goals started and hence the plans interacting increased, resulting in more plans not being used so more resources being

saved for use in achieving further goals. In the high level resource availability for the general tree, this lead to all goals being achieved by the Petri net model.

In the experiments for the deep tree, the Petri net timings even when including the loading times were quite similar to those for the constraint model, however in the experiments for the other two tree structures, especially the large sized general tree, the time taken for loading the Petri net model was greater than the time taken for the constraint model to find a solution. Despite the additional time taken for the reasoning in both models, the benefits gained from performing the reasoning over those shown in the random Petri net model show that it is worth considering taking the time to find a good solution. In dynamic environments, there may not be the time available to consider this as too much would have changed by the time a simulation had finished.



(a) Combined Timing (b) Combined Goals (c) Combined Plans

Figure 6.27: Comparison results for combined reasoning across the three tree structures

6.5.5 Conclusion

While the Petri net model has the faster running times, it also has the slowest loading times with the greatest memory usage once loaded. One of the side effects of this is that, as the size of the trees or the number of top-level goals increases, the load times rapidly increase until the application running the Petri net simulations is no longer able to load the Petri net goal-plan tree representation. Refinements and changes in the way the goals are represented may reduce the problem allowing greater numbers of goals to be handled in the Petri net model. Similarly, it is possible that refinements in the efficiency of the Prolog constraints used in the

constraint-based model may improve the performance of this model as well.

As has been shown by these experiments and discussed above, the constraint model produces the same results each time, with just a small variation in the time taken between repeats, while there is greater variance in the performance of the Petri net model. For an agent looking for unchanging results then the constraint model provides the better solution, however in some cases the Petri net model can give better results. In particular, when reasoning about positive interaction, the Petri net model gives better results on the number of plans used, and when reasoning about negative interference, the Petri net model also gives faster results for successfully achieving all goals, even when including the loading times. In the deep tree, both models perform well at reasoning about consumable resources, however when the reasoning is applied to the other two tree structures, the constraint model is able to achieve more goals when the resource availability is low.

Where the ability to reason about large number of goals is required, especially for large sized trees, the constraint model demonstrated that it was able to scale and find solutions to larger problems, however the trade-off comes at the time taken, taking 9 hours to reason about 75 goals.

The results presented here have compared the individual types of reasoning within the models and the combined reasoning. While in most cases it makes sense to combine all three types of reasoning, there may be application areas where only one is needed. For example, in applications where there is limited availability of consumable resources but very little interaction between the goals it may only make sense to use the resource reasoning. Similarly, in applications where there are a lot of common goals to achieve the same effects, and abundant resources it may be possible to just use the positive interaction reasoning. In applications where there is likely to be a lot of conflict between the goals or where it is more critical that all the goals are achieved, but again with abundant resources, it may be sufficient to just apply the negative interference reasoning.

In conclusion, the following recommendations can be made to agents about which model they may wish to consider:

- When just considering resource reasoning, if the goal-plan trees contain a lot

of branching then the constraint-based model gives better results in terms of goals achieved.

- When just considering positive interaction reasoning, the Petri net model gives better results for all goal-plan tree structures in terms of the reduction in plans used.
- When just considering negative interaction reasoning, the Petri net model gives better results for all goal-plan tree structures in terms of the time taken to perform the reasoning.
- When considering the combining of all three types of reasoning, the constraint-based model gives better results in terms of goals achieved except when there is high resource availability in which case the Petri net model performs better.
- When there are a large number of large goals (i.e., 50 or more goals containing more than 100 plans), the constraint-based model is able to perform the reasoning however it will take a long time to find a solution.

Chapter 7

Conclusions and Future Work

As technology is constantly evolving, the application of intelligent agents is becoming increasingly popular in a wide range of applications, in particular in dynamic environments where agents have incomplete knowledge of their surroundings. The agents are often required to pursue multiple goals simultaneously in these applications, so they need to be able to reason about the interactions between the goals and any constraints restricting the agent such as the limited availability of resources.

Summary of contributions In this thesis we have considered three domain independent types of reasoning about goals, represented using a goal-plan tree structure, that rational agents could apply when considering new goals to adopt and when selecting which plans to use in order to achieve the adopted goals. The three types of reasoning considered are: reasoning about the limited availability of consumable resources; positive interactions between goals; and negative interference between goals (see chapter 3).

These three types of reasoning have been incorporated into two models developed here for representing the goal-plan tree problem, the first is a Petri net model (see chapter 4) and the second is a constraint-based model (see chapter 5). These two models each represent a goal-plan tree with modules representing the different types of reasoning, that can be incorporated into them to perform the different types of reasoning. The three types of reasoning can either be used individually or combined together, the greatest increase in performance coming from

combining the reasoning about positive interactions between goals with that of reasoning about the limited availability of consumable resources. This is because the positive interaction reduces the number of plans used and hence the amount of resources being consumed, thereby allowing additional goals to be achieved with the resources saved.

When the models are executed, they produce outputs indicating the goals that can be safely adopted and the plans that can be used to achieve them. This is provided in the form of a list giving a suggested ordering in which the plans could be safely executed in order to avoid any interference. Each of the models is capable of reasoning effectively about the three types of reasoning considered in this thesis and, as shown in chapter 6, they both provide beneficial results, especially when compared to the performance without any reasoning included, as shown in the random (Petri net) model.

While both models perform effectively in each of the goal-plan tree structures evaluated here, it has been possible to identify tree structures where one approach is more successful than the other, or situations where it may be preferable to use one type of reasoning over the other. For example, the results for the broad tree scenario in section 6.3 show the constraint-based model performed better when reasoning about resources while the Petri net model performed better when reasoning about positive interactions. The number of plans saved by the Petri net model when reasoning about positive interaction in each of the tree structures is greater than that saved by the constraint-model due to differences between the two styles of implementation. However, the time taken to load the Petri net model gives a total time slightly greater than that taken by the constraint model. In each of the tree structures, the Petri net model provided overall better results when reasoning about negative interference. While both models were able to achieve all goals, the total time taken from loading to running a simulation in the Petri net was always less than the time taken to load and evaluate the constraints in the constraint-based model.

In the experiments, no time was allocated to the actual execution of the plans themselves so only the loading and reasoning times were recorded. If time was added for the execution of the plans, then it is possible that the loading time for the Petri net model and the evaluation time for the constraint-based model may

become negligible in comparison.

When considering the scalability of the two models, as was evaluated by the final tree structure in section 6.4, the constraint model performed considerably better than the Petri net model. The application used to simulate the Petri nets reached a limit where it was no longer able to load the large files required to represent the goal-plan trees, while the constraint model was able to load and evaluate the larger number of goals without any problems. However, the increase in time taken by the constraint model as the number of goals increased means that the use of this model in practice may be limited when considering large numbers of goals.

In situations where consistent good results are required and time is not an issue then the constraint satisfaction approach is shown to be a preferred model, particularly with a broad tree or with many large goal-plan trees. On the other hand, the Petri net model is more suited to deeper goal-plan trees, providing results quickly once loaded. As the results from the Petri net model vary slightly in each simulation, the short simulation time means that a number of repetitions can also be performed quickly with the Petri net approach. This allows the agent to potentially perform several repeats and select the best result based on its preferences, such as number of plans used or goals achieved where these can vary slightly between repeats of the positive reasoning and resource reasoning.

Comparison to existing approaches The types of reasoning that have been developed here are based on those defined in the work by Thangarajah *et al.* The relative performance and merits of the two approaches are discussed here.

The key difference between the approaches is the use of summary information in the different types of reasoning. In the reasoning about consumable resources, the summary information used by Thangarajah contained lists of *necessary* and *possible* resource requirements listing each of the resources and the quantity required. In the case of the two approaches developed here, the summary information defined focuses on *best case* and *worst case* resource requirements, since the aim of the two models is to select the best case wherever possible. When selecting which goals could be safely adopted, a normalised list of each of the types of resources and quantities required was used. However, for the subgoals with a choice

of plan branches, then a single number representing the sum of best case resource requirements for each branch was sufficient for selecting which plan branch to use. This means an overview of the resource requirements was stored for each of the top-level goals, with summed values being stored at the subgoal branches.

In the reasoning about the positive and negative interactions, the use of summary information was removed completely from the approaches developed here. For the positive interaction, this is because the reasoning here focuses on the effects being achieved by the plans, rather than synchronising the plans forcing them to wait until all matching plans are ready before selecting one to proceed. Here, the effects are achieved by the first available plan with the effects then ready to be used in the precondition of any further plans making use of the effects. In the Petri net model, checks are performed before attempting to execute a plan, to see if the effect has already been achieved, while in the constraint model, the duplicate plans are removed once identified so that only the plans that will be needed to achieve the goals are considered. This is possible as there was no negative interference included when just reasoning about positive interactions.

When reasoning about negative interference, the Petri net model protects all effects until they are no longer needed, while the constraint model sequences the plans identified as potentially interfering to ensure the interference is avoided. In both types of reasoning, once the interactions have been identified and dealt with, the information about the interactions is no longer required, removing the need for the summary information to be stored and monitored during execution.

To evaluate the performance of the approach developed by Thangarajah, two different tree structures were used. The first of depth 2 containing four plans and the second of depth 5 containing 12 plans. These were used to evaluate the performance of their approach when varying depths of tree, in particular for when reasoning about resources.

When comparing the goal-plan trees used in the evaluation of the approach developed by Thangarajah *et al.* and the evaluation here, there is a significant difference in the size of the trees. The experiments performed on the two models developed here were defined in a similar style to those adopted by Thangarajah and Padgham [2004], Thangarajah [2004]. This was to allow for a certain amount of comparison between the approaches. However, the goal structures used here are

considerably larger and more complex, so an exact comparison of performance is not possible. The specification of machines used to run the experiments is also different so time costs in particular will not be comparable.

In Thangarajah's approach and the approaches developed here, the evaluations compared the performance of the reasoning to a control case without any reasoning. Each case shows a similar improvement to those seen here over the control case, with only a small additional time cost for the reasoning models. When just considering the simulation times in Thangarajah's approach and the Petri net model, there is often a reduction in the time taken when compared to the control cases.

In the experiments by Thangarajah and Padgham [2004] for reasoning about resources, execution time for the plans themselves was included which increased the length of time for the simulation significantly. In comparison, their reasoning about negative interference and positive interaction simply focused on the reasoning cost without the additional plan execution time, with the negative interference being slower than the positive interaction reasoning. However, the time taken was less than the approaches developed here. In their approach, the average run time for the reasoning was 3 seconds, and without any reasoning just 1.2 seconds. Part of this increase is due to the extra number of goals being achieved, along with the computational costs introduced by the reasoning itself. While this run time is comparable to 10 medium sized broad tree goals when reasoning about negative interference in the Petri net model, the load times for the Petri net model are considerably more. The size of the trees used are also greater with at least twice as many plans in the small sized trees and four times the number of plans in the medium sized trees.

The results presented by Thangarajah *et al.* have evaluated each of the types of reasoning independently. While this has shown their individual effectiveness, a lot of the strength and practical application within agents involves the various types of reasoning being combined together as has been done here. Each of the types of reasoning produced in their work generated lists of summary information that was used in the reasoning process. As the size and complexity of the trees grow, these lists will also grow, potentially exponentially [Clement and Durfee, 2000a]. When combining the summary information for each of types of reasoning

together, this could lead to very large computational overheads, especially as the size of the trees increases. However, a lot of this computation is performed off-line so this increase may be acceptable when compared to the benefits gained from the reasoning.

The greatest benefit from combining the three types of reasoning comes from the positive interaction reasoning being combined with that about the limited availability of resources. As the positive interaction reasoning reduces the number of plans required for achieving some goals, the resource requirements for those goals lowers to the point where sufficient resources are saved for additional goals to be successfully completed.

Limitations While the approaches developed here are domain independent, they do have certain limitations. One of the current limitations to the approaches developed here is the assumption that plans always achieve their effects. Unfortunately in many application domains this assumption is not always applicable, especially in highly dynamic environments where there is a risk of plans failing.

Currently, in order for agents in an agent development language to make use of the reasoning developed here, they must first export their plan and goal descriptions to XML representations of the goal-plan trees. This can then be parsed to generate the desired models with the desired types of reasoning for the application. The developer must be able to specify the amount of resources that will be required by each plan if they wish to make use of the reasoning about consumable resources. The design of the reasoning has been based on the Belief Desire Intention (BDI) model of agents, so for agents that have been developed using different architectures it may not be as easy to generate the goal-plan tree representation required as input to the reasoning. However, a significant number of agent programming languages are based on the BDI model [Bordini *et al.*, 2005a, 2009].

Once the instances of the models for reasoning have been produced there is currently no automated method provided to alter them, for example where a completely new top-level goal is added, or where the requirements for an existing plan are changed. These changes would either need to be made manually, or new instances of the models generated. Particularly with the constraint satisfaction

based approach, where the reasoning process can take a long time, there is currently no mechanism built-in to allow reuse of partial solutions. Similarly, in the Petri net model, due to the length of time taken to load the goal representations, it may be desirable to modify the nets once loaded, rather than have to reproduce them when there is a change.

Future Work The reasoning about resources that has been considered here has focused on that of consumable resources that are limited in their availability. Another type of resource that is often used are reusable resources, such as communication channels. A model was shown in figure 4.6(a) of how this could be incorporated into the Petri net model, and constraints could be added into the constraint-based approach to prevent two plans attempting to use the same reusable resource at the same time. This was not initially included as the use of these resources can be scheduled, while the use of consumable resources has greater restrictions applied to it. The reasoning about consumable resources is also the more difficult of the two types of resources, with it being possible to later incorporate the reasoning for reusable resources easily. In addition, when considering consumable resources, all the goals are assumed to consume resources without any goals to recharge them. The Petri net approach and to some extent the constraint-based approach are however robust enough to handle this, at least in a simplistic manner. However, further work to extend both approaches to allow for more generic maintenance goals, as well as achievement goals is required.

The results on resource reasoning showed that the constraint-based model was generally better at handling the limited availability, allowing more goals to be achieved. This was due to the sequential nature of the constraints solvers and the ability to order the goals based on their resource requirements. While a benefit of Petri nets is their concurrent behaviour, incorporating a mechanism to control the order in which goals are adopted should improve the performance of the Petri net model when reasoning about resources.

When considering the effects caused by plans, it is assumed that all effects are reversible, allowing plans that could interfere with each other to be scheduled to achieve all goals. In some cases, the effects are permanent so this would not be possible. Incorporating this into the models would allow the reasoning to be used

in a greater number of applications.

A further extension that is required is to handle plan failures, which is particularly important for agents operating in highly dynamic environments. While plans can fail due to negative interference, properties of the environment can also change either due to other agents in the environment or natural environmental changes that occur over time. For agents to be truly rational and intelligent, they need to be able to handle plans failing and recover from such failures, rather than allowing them to cause the goal to fail.

When evaluating the two approaches, the Petri net editor failed to load the instances of the model for reasoning about large numbers of top-level goals with the final large goal-plan tree structure. In section 4.6, alternative approaches to representing the goals in the Petri net model were suggested. One or more of these alternatives can be tried to compare how well each of the approaches scales, along with loading time, against the approach used here. It is possible that refinements in both models will reduce either the loading times or the evaluation times providing improvements in the performance of both approaches.

Currently, when a developer is programming the agents, they will need to implement their own appropriate mechanisms for managing any possible conflicts between the goals. To make use of the reasoning developed here, they would need to export the goals and plans for the agent into the XML representation of goal-plan trees. From that point, they can generate the instances of the required model with the necessary types of reasoning incorporated, then manually apply the results to the agents' plan selection. By incorporating these approaches as an extension to an agent development language such as AgentSpeak, developers would be able to gain benefits from the improvements provided by the reasoning without having to manually include reasoning into every application developed.

Bibliography

H. L. Akin. Managing an autonomous robot team: The cerberus team case study. *International Journal of Human-friendly Welfare Robotic Systems*, 6(2):35–40, 2005.

A. Alshamsi, S. Abdallah; and I. Rahwan. Multiagent self-organization for a taxi dispatch system. In *proceedings of Eighth International Conference on Autonomous Agents and Multiagent Systems*, pages 21–28, Budapest, Hungry, May 2009. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS).

D. R. Anderson, D. J. Sweeney, T. A. Williams, J. Freeman; and E. Shoesmith. *Statistics for Business and Economics*. Thomson Learning, London, UK, 2007.

Q. Bai, M. Zhang; and K. T. Win. A colored petri net based approach for multi-agent interactions. In *proceedings of the 2nd IEEE International Conference on Autonomous Robots and Agents*, pages 152–157. ICARA, 2004.

I. Bakam, F. Kordon, C. L. Page; and F. Bousquet. Formalization of a spatialized multiagent model using coloured petri nets for the study of an hunting management system. In *FAABS '00: Proceedings of the First International Workshop on Formal Approaches to Agent-Based Systems-Revised Papers*, pages 123–132, London, UK, 2001. Springer-Verlag.

M. Baldoni, G. Boella; and L. van der Torre. Bridging agent theory and object orientation: Importing social roles in object oriented languages. In *Programming Multi-Agent Systems*, volume 3862 of *Lecture Notes in Computer Science*, pages 57–75. Springer, 2006.

M. Benisch and N. M. Sadeh. Examing DCSP coorination tradeoffs. Technical Report CMU-ISRI-05-140, Institute for Software Research International, Carnegie Mellon University, PA 15213, December 2005.

J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno; and M. Weber. The petri net markup language: Concepts, technology, and tools. In *proceedings of 24th International Conference on Applications and Theory of Petri Nets*, volume 2679 of *Lecture Notes in Computer Science*, pages 1023–1046, Eindhoven, The Netherlands, March 2003. Springer.

K. E. Björnberg. What relations can hold among goals, and why does it matter? *Crítica, Revista Hispanoamericana de Filosofía*, 41(121):47–66, April 2009.

O. Bonnet-Torrès and C. Tessier. From team plan to individual plans: a petri net-based approach. In *proceedings of AAMAS'05, 4th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 797–804, New York, July 2005. ACM Press.

S. Bonura, V. Morreale, G. Francavigilia, A. Marguglio, G. Cammarata; and M. Puccio. Intentions in bdi agents: From theory to implementation. In *7th International Conference on Practical Applications of Agents and Multi-Agent Systems*, volume 55/2009 of *Advances in Soft Computing*, pages 227–236. Springer, 2009.

R. H. Bordini, A. L. C. Bazzan, R. de Oliveira Jannone, D. M. Basso, R. M. Viccari; and V. R. Lesser. AgentSpeak(XL): Efficient intention selection in BDI agents via decision-theoretic task scheduling. In C. Castelfranchi and W. Johnson, editors, *proceedings of First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2002)*, pages 1294–1302, New York, USA, July 2002. NY: ACM Press.

R. H. Bordini, M. Dastani, J. Dix; and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Number 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer-Verlag, 2005a.

R. H. Bordini, J. F. Hübner; and R. Vieira. ***Jason*** and the golden fleece of agent-oriented programming. In *Multi-Agent Programming*. Springer, 2005b.

R. H. Bordini, J. F. Hübner; and M. J. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using* Jason. Wiley Series in Agent Technology. John Wiley and Sons, Ltd., 2007.

R. H. Bordini, M. Dastani, J. Dix; and A. E. F. Seghrouchni, editors. *Multi-Agent Programming: Languages, Tools and Applications*. Spriger-Verlag, 2009.

I. Bratko. *Prolog: Programming for Artificial Intelligence*. International Computer Science Series. Pearson Education Limited, third edition, 2001.

M. E. Bratman. What is intention? In P. R. Cohen, J. Morgan; and M. E. Pollack, editors, *Intentions in Communication*, chapter 2, pages 15–32. MIT Press, June 1990.

P. Busetta, R. Rönnquist, A. Hodgson; and A. Lucas. JACK intelligent agents - components for intelligent agents in java. *AgentLink, Available from http://www.agentlink.org/newsletter/2/newsletter2.pdf*, 2:2–5, 1999.

C. Castelfranchi and R. Falcone. Conflicts within and for collaboration. In C. Tessier, L. Chaudron; and H.-J. Müller, editors, *Conflicting Agents: Conflict Management in Multiagent Systems*, Multiagent systems, Artificial societies, and Simulated organizations, chapter 2, pages 33–62. Kluwer Academic Publishers, 2001.

L. Ceccaroni and D. Robertson. WaRP - a reactive planner integrated in an environmental decision-support system for wastewater treatment plant management. In *ECAI 2000: 14th European Conference on Artificial Intelligence*, volume 14, pages 491–495, August 2000.

A. Cheadle. The ECLiPSe constraint programming system, September 2008. URL `http://www.eclipse-clp.org/eclipse`.

Y. Chen, B. W. Wah; and C.-W. Hsu. Temporal planning using subgoal partitioning and resolution in SGPlan. *Journal of Artificial Intelligence Research*, 26: 323–369, 2006.

C.-C. Cheng and S. F. Smith. Applying constraint satisfaction techniques to job shop scheduling. Technical Report CMU-RI-TR-95-03, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, January 1995.

S. Christensen and N. D. Hansen. Coloured petri nets extended with place capacities, test arcs and inhibitor arcs. *Lecture notes in Computer Science*, 691: 186–205, 1993.

B. Clement, E. Durfee; and A. Barrett. Abstract reasoning for planning and coordination. *Lecture notes in Computer Science*, Jan 2002.

B. J. Clement and E. H. Durfee. Identifying and resolving conflicts among agents with hierarchical plans. In *proceedings of AAAI Workshop on Negotiation: Settling Conflicts and Identifying Opportunities, Technical Report WS-99-12*, pages 6–11. AAAI Press, 1999a.

B. J. Clement and E. H. Durfee. Exploiting domain knowledge with a concurrent hierarchical planner. In *proceedings of AI and Planning Systems (AIPS-2000) Workshop on Analysing and Exploiting Domain Knowledge for Efficient Planning*, pages 57–62, April 2000a. Working Notes.

B. J. Clement and E. H. Durfee. Theory for coordinating concurrent hierarchical planning agents using summary information. In *AAAI '99/IAAI '99: Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence*, pages 495–502, Menlo Park, CA, USA, 1999b. American Association for Artificial Intelligence.

B. J. Clement and E. H. Durfee. Performance of coordinating concurrent hierarchical planning agents using summary information. In *proceedings of 4th International Conference on Multi-Agent Systems (ICMAS)*, pages 373–374, Boston, Massachusetts, USA, July 2000b. IEEE Computer Society.

B. J. Clement, A. C. Barrett, G. Rabideau; and E. H. Durfee. Using abstraction in planning and scheduling. In *proceedings of the Sixth European Conference on Planning (ECP-01)*, September 2001.

P. Codognet and D. Diaz. Compiling constraint in clp(FD). *Journal of Logic Programming*, 27(33):185–226, 1996.

C. Conway, C.-H. Li; and M. Pengelly. Pencil: A petri net specification language for java, December 2002.

R. S. Cost, Y. Chen, T. Finin, Y. Labrou; and Y. Peng. Modeling agent conversations with colored petri nets. In *proceedings of Working notes of the Autonomous Agents '99 Workshop on Specifying and Implementing Conversation Policies*, Seattle, Washington, May 1999.

R. S. Cost, Y. Chen, T. Finin, Y. Labrou; and Y. Peng. Using colored petri nets for conversation modeling. In *Agent Communication Languages*, volume 1916 of *Lecture Notes in AI*, pages 178–192. Springer-Verlag, 2000.

M. Dastani. 2apl: A practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, June 2008.

L. de Silva, S. Sardina; and L. Padgham. First principles planning in bdi systems. In *proceedings of Eighth International Conference on Autonomous Agents and Multiagent Systems*, pages 1105–1112, Budapest, Hungry, May 2009. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS).

R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

D. Diaz. The GNU prolog web site, February 2009. URL `http://www.gprolog.org`.

D. Diaz and P. Codognet. The GNU prolog system and its implementation. In *proceedings of the 2000 ACM Symposium on Applied Computing*, volume 2, pages 728–732. ACM Press, 2000.

M. D'Inverno, M. Luck, M. P. Georgeff, D. Kinny; and M. J. Wooldridge. The dMARS architecture: A specification of the distributed multi-agent reasoning system. *Autonomous Agents and Multi-Agent Systems*, 9(1-2):5–53, October 2004.

D. A. Dolgov and E. H. Durfee. Resource allocation among agents with mdp-induced preferences. *Journal of Artificial Intelligence (JAIR)*, 27:505–549, December 2006.

M. Duvigneau, D. Moldt; and H. Rölke. Concurrent architecture for a multi-agent platform. In F. Giunchiglia, J. Odell; and G. Weiß, editors, *proceedings of Agent-Oriented Software Engineering III. Third International Workshop, Agent-oriented Software Engineering (AOSE)2002, Bologna, Italy, July 2002. Revised Papers and Invited Contributions*, volume 2585, pages 59–72. Springer, LNCS, 2003.

K. Erol, J. Hendler; and D. S. Nau. Semantics for hierarchical task-network planning. Technical Report 64432, King Fahd University of Petroleum and Minerals, 1994.

T. A. Estlin, A. Gray, T. Mann, G. Rabideau, R. Castano, S. Chien; and E. Mjolsness. An integrated system for multi-rover scientific exploration. In *proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI)*, pages 613–620, July 1999.

FIPA. Specification part 2 - agent communication language. Technical report, The Foundation for Intelligent Physical Agents, 1999.

M. Fisher and C. Ghidini. Exploring the future with resource-bounded agents. *Journal of Logic, Language and Information*, 18(1):3–21, January 2009.

M. I. A. Galipienso and F. B. Sanchís. A mixed closure-CSP method to solve scheduling problems. In *IEA/AIE '01: Proceedings of the 14th International conference on Industrial and engineering applications of artificial intelligence and expert systems*, pages 559–570, London, UK, 2001. Springer-Verlag.

M. Georgeff, B. Pell, M. Pollack, M. Tambe; and M. Wooldridge. The belief-desire-intention model of agency. In J. P. Müller, M. P. Singh; and A. S. Rao, editors, *Intelligent Agents V—Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98), held as part of the Agents' World, Paris, 4–7 July, 1998*, number 1555 in Lecture Notes in Artificial Intelligence, pages 1–10, Heidelberg, 1999. Spriger-Verlag.

M. P. Georgeff. Communication and interaction in multi-agent planning. In *proceedings of American Association for Artificial Intelligence*, 1983.

M. P. Georgeff and A. L. Lansky. Procedural knowledge. *Proceedings of the IEEE Special issue on Knowledge Representation*, 74(10):1383–1398, October 1986.

M. Hannebauer. Their problems are my problems - the transition between internal and external conflict. In C. Tessier, L. Chaudron; and H.-J. Müller, editors, *Conflicting Agents: Conflict Management in Multiagent Systems*, Multiagent systems, Artificial societies, and Simulated organizations, chapter 3, pages 63–110. Kluwer Academic Publishers, 2001.

M. Hannebauer. A formalization of autonomous dynamic reconfiguration in distributed constraint satisfaction. *Fundamenta Informaticae*, 43(1–4):129–151, 2000.

J. F. Horty and M. E. Pollack. Evaluating new options in the context of existing plans. *Artificial Intelligence*, 127(2):199–220, 2004.

M. J. Huber. JAM: a BDI-theoretic mobile agent architecture. In *proceedings of the Third International Conference on Autonomous Agents*, pages 236–243, Seattle, Washington, USA, 1999. ACM Press.

M. Jakob, M. Pěchouček, S. Miles; and M. Luck. Case studies for contract-based systems. In *proceedings of seventh International Conference on Autonomous Agents and Multiagent Systems*, pages 55–62. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), May 2008.

N. R. Jennings, K. Sycara; and M. Wooldridge. A roadmap of agent research and development. *International Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

M. Köhler and H. Rölke. Properties of object petri nets. In *Applications and Theory of Petri Nets 2004*, volume 3099 of *Lecture notes in Computer Science*, pages 278–297. Springer, 2004.

L. M. Kristensen, S. Christensen; and K. Jensen. The practitioner's guide to coloured petri nets. *International Journal on Software Tools for Technology Transfer: Special section on coloured Petri nets*, 2(2):98–132, 1998.

M. Kumar and S. Rajotia. Integration of process planning and scheduling in a job shop environment. *International Journal of Advanced Manufacturing Technology*, 28(1-2):109–116, February 2006.

O. Kummer, F. Wienberg; and M. Duvigneau. Renew – the Reference Net Workshop, May 2006. URL `http://www.renew.de/`. Release 2.1.

J. J. Leifer and R. Milner. Transition systems, link graphs and petri nets. Technical Report UCAM-CL-TR-598, Computer Laboratory, University of Cambridge, Cambridge, UK, August 2004.

J. Leite, J. Alferes; and B. Mito. Resource allocation with answer-set programming. In *proceedings of Eighth International Conference on Autonomous Agents and Multiagent Systems*, pages 649–656, Budapest, Hungry, May 2009. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS).

V. Lesser, B. Horling, F. Klassner, A. Raja, T. Wagner; and S. X. Zhang. BIG: A resource-bounded information gathering and decision support agent. *Artificial Intelligence*, 118(1-2):197–244, January 2000.

R. T. Maheswaran, M. Tambe, E. Bowring, J. P. Pearce; and P. Varakantham. Taking DCOP to the real world : Efficient complete solutions for distributed event scheduling. In *proceedings of Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1 (AAMAS'04)*, pages 310–317. AAMAS, 2004.

R. Mailler and V. Lesser. Solving distributed constraint optimization problems using cooperative mediation. In *proceedings of Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, pages 438–445. IEEE Computer Society, 2004a.

R. Mailler and V. Lesser. Using cooperative mediation to solve distributed constraint satisfaction problems. In *proceedings of Third International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS 2004)*, pages 446–453. IEEE Computer Society, 2004b.

R. Mailler and V. Lesser. Asynchronous partial overlay: A new algorithm for solving distributed constraint satisfaction problems. *In Journal of Artificial Intelligence Research (JAIR)*, 25:529–576, 2006.

L. Matthies, E. Gat, R. Harrison, B. Wilcox, R. Volpe; and T. Litwin. Mars microrover navigation: Performance evaluation and enhancement. *Special Issue on Autonomous Vehicles for Planetary Exploration*, 2(4):291–311, 1995.

H. Mazouzi, A. E. F. Seghrouchni; and S. Haddad. Open protocol design for complex interactions in multi-agent systems. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 517–526, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-480-0. doi: http://doi.acm.org/10.1145/544862.544866.

F. Meneguzzi and M. Luck. Motivations as an abstraction of meta-level reasoning. In *Multi-Agent Systems and Applications V*, volume 4696/2007 of *Lecture Notes in Computer Science*, pages 204–214. Springer, 2007.

T. Mora, A. Sesay, J. Denzinger, H. Golshan, G. Poissant; and C. Konecnik. Cooperative search for optimizing pipeline operations. In *proceedings of seventh International Conference on Autonomous Agents and Multiagent Systems*, pages 115–122, Estoril, Portugal, May 2008. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS).

T. Murata. Petri nets: Properties, analysis and applications. In *proceedings of the IEEE, Vol. 77, Issue 4*, pages 541–580. IEEE, April 1989.

N. Muscettola, P. P. Nayak, B. Pell; and B. C. Williams. Remote agent: to boldly go where no ai system has gone before. *Artificial Intelligence*, 103(1-2):5–47, August 1998.

R. Nair, P. Varakantham, M. Tambe; and M. Yokoo. Networked distributed POMDPs: A synergy of distributed constraint optimization and POMDPs. In *proceedings of Twentieth National Conference on Artificial Intelligence (AAAI-05)*, pages 1758–1760, 2005.

T. J. Norman, A. Preece, S. Chalmers, N. R. Jennings, M. Luck, V. D. Dang, T. D. Nguyen, V. Deora, J. Shao, W. A. Gray; and N. J. Fiddian. CONOISE: Agent-based formation of virtual organisations. In *proceedings of 23rd SGAI International Conference on Innovative Techniques and Applications of AI Special Issue of Knowledge Based Systems*, pages 353–366. Cambridge, UK, 2003.

I. Nourbakhsh, K. Sycara, M. Koes, M. Yong, M. Lewis; and S. Burion. Human-robot teaming for search and rescue. *IEEE Pervasive Computing: Mobile and Ubiquitous Systems*, pages 72–78, January 2005.

J. Odell. Objects and agents compared. *Journal of Object Technology*, 1(1):41–53, May-June 2002.

L. Padgham and M. Winikoff. *Developing Intelligent Agent Systems: A practical guide*. John Wiley and Sons, Ltd., June 2004.

J. Palmer. Autonomous tech 'requires debate', September 2009. URL `http://news.bbc.co.uk/1/hi/technology/8210477.stm`.

P. Paruchuri, E. Bowring, R. Nair, J. P. Pearce, N. Schurr, M. Tambe; and P. Varakantham. Multiagent teamwork: Hybrid approaches. In *proceedings of Computer society of India Communications*, 2006.

J. P. Pearce, R. T. Maheswaran; and M. Tambe. Solution sets in DCOPs and graphical games. In *proceedings of Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 577–584, Hakodate, Japan, May 2006. ACM Press.

J. L. Peterson. *Petri Net Theory and the modeling of Systems*. Prentice-Hall, 1981.

A. Pokahr, L. Braubach; and W. Lamersdorf. A goal deliberation strategy for BDI agent systems. In *Third German conference on Multi-Agent System TEchnologieS (MATES-2005); Springer-Verlag, Berlin Heidelberg New York, pp. 82-94*. Springer-Verlag, Berlin Heidelberg New York, September 2005.

A. Raja and V. Lesser. Reasoning about coordination costs in resource-bounded multi-agent systems. *proceedings of AAAI 2004 Spring Symposium on Bridging the multiagent and multi robotic research gap*, pages 25–40, March 2004a.

A. Raja and V. Lesser. Meta-level reasoning in deliberative agents. In *proceedings of international conference on Intelligent Agent Technology*, pages 141–147, September 2004b.

P. Ramachandran and M. Kamath. A sufficient condition for reachability in a general petri net. *Discrete Event Dynamic Systems: Theory and Applications*, 14(3):251–266, July 2004.

A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In R. van Hoe, editor, *proceedings of Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Eindhoven, The Netherlands, 1996.

A. S. Rao and M. P. Georgeff. Bdi agents: From theory to practice. Technical Note 56, Australian Artificial Intelligence Institute, 1995.

S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, chapter 17: Making Complex Decisions, pages 613–648. Prentice Hall, New Jersey, 2nd edition, 2003.

S. Sardina, L. de Silva; and L. Padgham. Hierarchical planning in BDI agent programming. In *proceedings of Fifth International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1001–1008, Hakodate, Japan, May 2006.

P. Scerri, D. V. Pynadath; and M. Tambe. Towards adjustable autonomy for the real world. *Journal of Artificial Intelligence Research*, 17:171–228, 2002.

N. Schurr, J. Marecki, J. P. Lewis, M. Tambe; and P. Scerri. The defacto system: Coordinating human-agent teams for the future of disaster response. In *Multi-Agent Programming*. Springer, 2005.

J. R. Searle. *Speech Acts: an Essay in the Philosophy of Language*. Cambridge University Press, 1969.

A. E. F. Seghrouchni and S. Haddad. A recursive model for distributed planning. In *proceedings of Second International Conference on Multi-Agent Systems (ICMAS)*. IEEE press, December 1996.

P. Shaw and R. Bordini. Towards alternative approaches to reasoning about goals. In *Proceedings of the 5th International Workshop on Declarative Agent Languages and Technologies*, volume 4897/2008 of *Lecture Notes in Computer Science*, pages 104–121. Springer, January 2008.

P. Shaw, B. Farwer; and R. H. Bordini. Theoretical and experimental results on the goal-plan tree problem. *Proceedings of the 7th international joint conference on Autonomous Agents and Multiagent Systems*, 3:1379–1382, May 2008.

G. Simari and S. Parsons. On the relationship between MDPs and the BDI architecture. In *proceeding of Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'06)*, pages 1041–1048, May 2006.

L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press series in logic programming. MIT Press, second edition, 1994.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

M. Tambe, E. Bowring, H. Jung, G. Kaminka, R. T. Maheswaran, J. Marecki, P. J. Modi, R. Nair, S. Okamoto, J. P. Pearce, P. Paruchuri, D. Pynadath, P. Scerri, N. Schurr; and P. Varakantham. Conflicts in teamwork: Hybrids to the rescue. In *proceedings of the Fourth International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 3–12, New York, 2005. ACM Press.

C. C. Teo, R. Bhatnagar; and S. C. Graves. An extension to the tactical planning model for a job shop: Continuous-time control. In *proceeding of Singapore-MIT Alliance (SMA) Conference*, pages 8–16, July 2005.

C. Tessier, L. Chaudron; and H.-J. Müller, editors. *Conflicting Agents: Conflict Management in Multiagent Systems*. Multiagent systems, Artificial societies, and Simulated organizations. Kluwer Academic Publishers, 2001.

J. Thangarajah. *Managing the Concurrent Execution of Goals in Intelligent Agents*. PhD thesis, School of Computer Science and Informaiton Technology, RMIT University, Melbourne, Victoria, Australia, December 2004.

J. Thangarajah and L. Padgham. An empirical evaluation of reasoning about resource conflicts in intelligent agents. In *proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 04)*, pages 1298–1299, 2004.

J. Thangarajah, M. Winikoff; and L. Padgham. Avoiding resource conflicts in intelligent agents. In *proceedings of 15th European Conference on Artifical Intelligence 2002 (ECAI 2002)*, Amsterdam, 2002. IOS Press.

J. Thangarajah, L. Padgham; and M. Winikoff. Detecting and avoiding interference between goals in intelligent agents. In *proceedings of 18th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 721–726, Acapulco, Mexico, August 2003a. Morgan Kaufmann.

J. Thangarajah, L. Padgham; and M. Winikoff. Detecting and exploiting positive goal interaction in intelligent agents. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 401–408, New York, NY, USA, 2003b. ACM Press.

J. Thangarajah, J. Harland; and N. Yorke-Smith. A soft COP model for goal deliberation in a BDI agent. *Sixth International Workshop on Constraint Modelling and Reformulation (ModRef'07) on Constraint Modelling and Reformulation (ModRef'07)*, 2007.

The RoboCup Federation. RoboCup rescue official web page, 2009a. URL `http://www.robocuprescue.org/`.

The RoboCup Federation. Robocup official site, July 2009b. URL `http://www.robocup.org/`.

Theorectical Foundations Group. Renew - the reference net workshop, 2006. URL `http://www.renew.de/`. Department of Informatics, University of Hamburg.

N. A. M. Tinnemeier, M. Dastani; and J.-J. C. Meyer. Goal selection strategies for rational agents. In *Languages, Methodologies and Development Tools for Multi-Agent Systems*, volume 5118/2008 of *Lecture Notes in Computer Science*, pages 54–70. Springer, 2008.

W. Truszkowski, M. Hinchey, J. Rash; and C. Rouff. Autonomous and autonomic systems: a paradigm for future space exploration missions. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 36(3): 279–291, May 2006.

J. Tsai, S. Rathi, C. Kiekintveld, F. Ordóñez; and M. Tambe. IRIS - a tool for strategic security allocation in transportation networks. In *proceedings of Eighth International Conference on Autonomous Agents and Multiagent Systems*, pages 37–44, Budapest, Hungry, May 2009. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS).

E. Tsang. *Foundation of Constraint Satisfaction*. Academic Press, 1993.

P. Varakantham, R. T. Maheswaran; and M. Tambe. Practical POMDPs for personal assistant domains. In D. Shapiro, P. Berry, J. Gersh; and N. Schurr, editors, *proceedings of AAAI Spring Symposium*, Menlo Park, California, 2005. AAAI Press.

A. Walczak, L. Braubach, A. Pokahr; and W. Lamersdorf. Augmenting BDI agents with deliberative planning techniques. In *proceedings of Fourth International Workshop on Programming Mulit-Agent Systems (PROMAS)*, volume 4411/2007 of *Lecture Notes in Computer Science*, pages 113–127, Hakodate, Japan, May 2007. Springer-Verlag.

R. Washington, K. Golden, J. Bresina, D. E. Smith, C. Anderson; and T. Smith. Autonomous rovers for mars exploration. In *proceedings of the 1999 IEEE Aerospace Conference*, volume 1, pages 237–251. IEEE, March 1999.

R. Weigel and B. Faltings. Compiling constraint satisfaction problems. *Artificial Intelligence*, 115(2):257–287, 1999.

G. Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 1999.

D. Weyns and T. Holvoet. A colored petri net for a multi-agent application. In *proceedings of Modeling Components, Objects and Agents, MOCA'02*, pages 121–140. MOCA, August 2002.

J. Wielemaker. An overview of the SWI-prolog programming environment. In F. Mesnard and A. Serebrenik, editors, *proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, 2003.

M. Winikoff, L. Padgham, J. Harland; and J. Thangarajah. Declarative and procedural goals in intelligent agent systems. In *proceedings of Eighth International Conference on Principles of Knowledge*, pages 470–481, April 2002.

M. Wooldridge and N. Jennings. Intelligent agents: Theory and practice. *In Knowledge Engineering Review*, 10(2):115–152, January 1995.

M. J. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley and Sons, Ltd., 2nd edition, 2009.

M. Yokoo and K. Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, June 2000.

V. A. Ziparo and L. Iocchi. Petri net plans. In *proceedings of Fourth International Workshop on Modelling of Objects, Components, and Agents (MOCA)*, pages 267–290, 2006.

V. A. Ziparo, L. Iocchi, D. Nardi, P. Palamara; and H. Costelha. Petri net plans: A formal model for representation and execution of multi-robot plans. In *proceedings of the Seventh International Joint Conference on Autonomous Agents and*

Multiagent Systems, volume 1, pages 79–86, Estoril, Portugal, May 2008. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS).